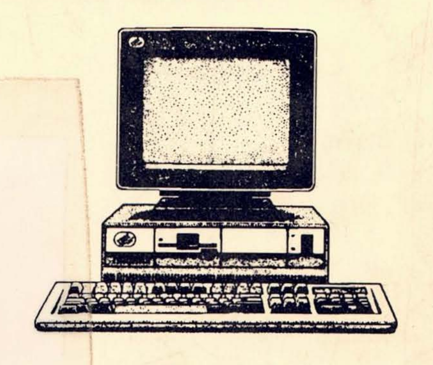
Manfred Moser

8086/8088

Assembler für Einsteiger

Einführung in die Programmierung der 8086/8088 mit Assembler







ISBN 3-88963-282-3

1. Auflage 1990

Gedruckt in der Bundesrepublik Deutschland - Printed in West-Germany - Imprimé en RFA.

Lesen Sie diesen Text, bevor Sie beginnen!

Die im Buch beschriebenen Programme und die dazugehörige Beschreibung wurden vom Autor und vom Verlag mit großer Sorgfalt erstellt und geprüft. Trotzdem lassen sich Fehler nie ganz vermeiden. Der HOFACKER-Verlag sieht sich deshalb gezwungen, weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für eventuelle Folgeschäden, die auf Programmfehler oder fehlerhafte Angaben in der Anleitung zurückzuführen sind, zu übernehmen. Bei eventuellen Fehlern sind wir für jeden schriftlichen Hinweis dankbar.

Eine Weitergabe an dritte Personen ist grundsätzlich verboten. Auch die unentgeltliche Weitergabe führt zu Schadensersatzansprüchen des Verlages.

Assembler
für
Einsteiger

Einführung in die Programmierung des 8086/8088 mit Assembler

Manfred Moser

Inhaltsverzeichnis

I. Zur Einführung

1. Einige Grundbegriffe

Der zentrale Bauteil eines Computers, der Prozessor, besteht vorwiegend aus (elektronischen) Schaltern, die zwei Stellungen einnehmen können: ein - aus, oder auch: links - rechts, oben - unten usw. Formal bezeichnet man einen solchen Schalter als "Bit" (BInary digiT), seine beiden Stellungen mit "0" bzw. "1", man sagt: das Bit "hat den Wert" 0 bzw. 1. Jeder Schalter kann durch einen elektrischen Impuls in Stellung "0" bzw. in Stellung "1" gebracht werden. Jeweils eine Gruppe von solchen Schaltern ist zu einer größeren Einheit zusammengefaßt; bei den Prozessoren, die ihre Hersteller mit 8086 bzw. 8088 bezeichnen, sind dies im allgemeinen 16 Schalter - also sind dies "16-Bit-Prozessoren". Schreibt man die Stellung der einzelnen Schalter einer solchen Einheit der Reihe nach hintereinander an, z.B.

1111011111010000,

ergibt sich das Bild einer Zahl des binären Zahlensystems (die gleich 63440 des "normalen" dezimalen Zahlensystems ist). Dieses Bild ist ziemlich unübersichtlich; so hat sich eingebürgert, jeweils vier Ziffern zusammenzufassen, also

1111 0111 1101 0000

zu schreiben, und für jede dieser kleineren Gruppen die entsprechende Ziffer des hexadezimalen Zahlensystems (= "16er System"; "hex") - siehe dazu den Abschnitt VIII - zu setzen, also

F 7 D 0.

Bei der Numerierung der Schalter, der "Bits", in der Einheit beginnt man von rechts mit Nummer 0. Daß im ganz links stehenden Bit mit der Nummer 15 im Beispiel oben eine "1" steht, be-

deutet unmittelbar nicht eine "Zahl" oder etwas ähnliches, sondern schlicht und einfach nur, daß der Schalter Nr. 15 in der gemeinten Einheit in Stellung "1" ist.

Die weitaus größere Anzahl solcher Einheiten hat nur die "Aufgabe", die einmal eingegebene Stellung der einzelnen Schalter festzuhalten, zu "speichern", bis sie durch äußere Umschalt-Impulse geändert wird - dies sind "Speicher"-Einheiten. Eine kleine Anzahl weiterer Einheiten hat viel mehr zu arbeiten, sie hat die "Aufgabe", die Umschalt-Impulse "richtig" durch den Prozessor (und den gesamten Computer) zu leiten - dies geschieht durch die jeweilige Kombination der Schalterstellungen dieser "Schalt"-Einheiten.

Dazu ein Beispiel:

Eine bestimmte **Speicher**einheit im Prozessor hat den Namen AX. Die 16 Bits dieser Einheit sollen folgende Stellung haben:

1010 0110 1100 0011

Die gerade wirksame "**Schalt**"-Einheit soll die oben angegebene Stellung ihrer Schalter haben (F7D0). Wenn nun in diese "Schalt"-Einheit ein "Arbeitsimpuls" gegeben wird, so wird dieser durch die Stellung ihrer einzelnen Schalter so geleitet, daß er in der Einheit AX die Stellungen aller Schalter gerade umkehrt. Nach dem Arbeitsimpuls haben die Bits in AX folgende Werte:

0101 1001 0011 1100

Die jeweilige Kombination der Schalterstellungen in der wirksamen "Schalt"-Einheit nennt man naheliegenderweise einen **Befehl**, der dafür verantwortlich ist, **was mit welchen** "Speicher"-Einheiten geschehen soll.

Der **Befehl** F7D0 - s.o. - bedeutet, daß alle Bits in der "Speicher"-Einheit mit dem Namen AX den gegenteiligen Wert erhalten sollen. Der ein wenig anders lautende Befehl F7D1 bedeutet das gleiche für die "Speicher"-Einheit mit dem Namen CX.

Die Kombination der Bit-Werte in den "Speicher"-Einheiten bedeutet **für sich genommen** nichts - auch wenn die jeweilige Kom-

bination eindeutig durch eine binäre bzw. hexadezimale Zahl dargestellt werden kann. Nur von den darauf wirkenden Befehlen hängt es ab, ob die Kombination eben als Zahl des binären Zahlensystems, mit der gerechnet werden soll, als Nummer einer anderen "Speicher"-Einheit, mit der etwas geschehen soll, oder als Code für ein Zeichen - oder, wenn diese Kombination durch einen entsprechenden Befehl in die "Schalt"-Einheit übertragen wird, sogar als Befehl behandelt oder interpretiert wird. So bedeuten die acht Bit 0100 0001 (41_{hex}) als **Zeichen** interpretiert (z.B. bei der Ausgabe auf dem Bildschirm) den Buchstaben "A", als **Zahl** bzw. als **Nummer** interpretiert "fünfundsechzig" (die Zahl bzw. Nummer, die dezimal 65 geschrieben wird) - und, wenn diese acht Bit in die "Schalt"-Einheit gelangen, also als **Befehl** interpretiert, die Vorschrift, die Schalterstellung in AX als Zahl zu interpretieren und 1 zu addieren.

"Befehle" des Computers sind also nichts anderes, als bestimmte Kombinationen von Schalterstellungen in den "Schalt"-Einheiten des Prozessors. Wenn also dem Computer ein bestimmter Befehl gegeben werden soll, muß dafür gesorgt werden, daß die einzelnen Schalter der "Schalt"-Einheiten "richtig" gestellt werden. Im Prinzip geschieht dies dadurch, daß die jeweils gewünschte Schalterstellung in einem "Programm" hintereinander angeschrieben wird, das der Computer auf Wunsch Befehl für Befehl "einliest", d.h. in die "Schalt"-Einheiten überträgt, und ausführt. Konkret heißt dies, daß im Programm festgelegt sein muß, daß zunächst Schalter Nr. 15 der "Schalt"-Einheit z.B. auf "1" gestellt wird, Nr. 14 auf "0" usw.; d.h.: der Prozessor "versteht" nur "0" und "1", "versteht" nur Befehle in der Darstellung von Binärzahlen oder, wie man üblicherweise sagt, Befehle im Binärcode. Es ist schon ein "Entgegenkommen" des Computers, daß er auch die Darstellung im Hexadezimalcode annimmt, der wegen der Zusammenfassung von vier Bits zu einer Hexadezimalziffer kürzer zu schreiben ist - er selbst "zerlegt" diese Ziffern wieder in die vier für den Prozessor verständlichen Bits.

Ohne weitere Hilfsmittel hätte also der Programmierer die Aufgabe, die von ihm gewünschte Befehlsfolge im Hexadezimalcode dem Computer mitzuteilen, der "Sprache", die dieser auch "versteht", die deswegen auch "Maschinensprache" genannt wird. Dies ist kein unmögliches Unterfangen, aber wer verbindet z.B. so ohne weiteres die Hexadezimalzahl 91 mit der Absicht, vom Computer zu verlangen, er soll die Schalter-Stellungen - den "Inhalt" - der beiden "Speicher"-Einheiten mit den Namen AX und CX gerade austauschen? Oder, wie oben angeführt, den Befehl, die Werte der Bits von AX sollten in ihr Gegenteil vertauscht werden, mit der Zahl F7D0? Schon diese beiden Beispiele deuten die mannigfaltige Verschiedenheit des Hexadezimalcodes für Befehle an; extrem zeigt sich diese im Befehl, den Inhalt einer "Speicher"-Einheit in eine andere zu kopieren: je nachdem, welche Einheiten betroffen sein sollen, bzw. auch je nachdem, was da kopiert werden soll, kann dieser Befehl mit den Hexadezimalzahlen 88, 89, 8A, 8B, 8C, 8E, A0, A1, A2, A3, B1, B2, B3, B4, B5, B6, B7, B8, B9, BA, BB, BC, BD, BE, BF, C6, C7 beginnen und zwischen vier und zwölf Hexadezimalziffern lang sein!

Hier erleichtert ein **Assembler** die Arbeit des Programmierers wesentlich: Ein Assembler ist ein - zunächst in der Maschinensprache geschriebenes - Programm, das die einzelnen Befehle, die der Programmierer in einer leichter verständlichen Form geschrieben hat, in die Maschinensprache übersetzt; diese "leichter verständliche Form" heißt "Assemblersprache" (kurz oft auch nur "Assembler", was manchmal Verwirrung stiften kann!): in ihr werden die Befehle in einleuchtenden Abkürzungen dargestellt: zunächst das, **was** geschehen soll, in Abkürzungen der (englischen) Alltagssprache - so z.B. für alle oben angeführten Kopier-Befehle das **eine** "Wort" MOV (von engl. move); dann das, **womit** dies geschehen soll, mit entsprechenden "Namen" der betroffenen "Speicher"-Einheiten. Dabei entspricht jedem möglichen Maschinenbefehl genau ein Befehl in der Assemblersprache. Wegen dieser "Eins-zu-eins-Übersetzung" der Befehle ist also ein Pro-

gramm in dieser Sprache noch sehr nahe an der Maschinensprache, der Assembler erlaubt ein "maschinennahes" und doch in seinen Einzelheiten leicht verständliches Programmieren.

Es stehen verschiedene Formen von Assemblern zur Verfügung: einmal der **Assembler** in DEBUG, einem Hilfsprogramm von DOS, der letztlich nur zur Herstellung von kurzen und kurzlebigen Hilfsprogrammen geeignet ist, und verschiedene **Makro-Assembler**, die noch unterschiedlich viele zusätzliche Hilfsmittel für das Programmieren zur Verfügung stellen, so daß sich einige davon fast nicht mehr von den sogenannten "Höheren Programmiersprachen" wie BASIC unterscheiden.

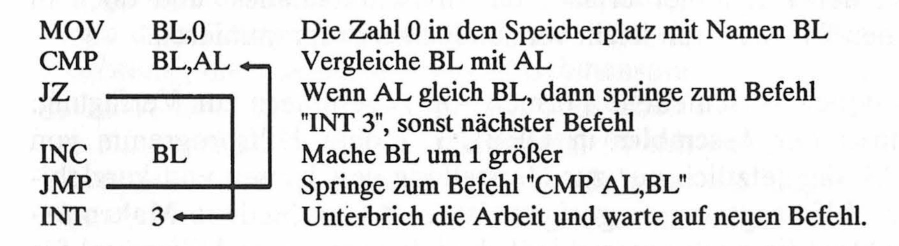
2. Die Problemstellung

Der hier vorgestellte Assembler versucht eine Schwierigkeit zu überwinden, die vor allem "Einsteiger" bei maschinennaher Programmierung haben können:

Wird auf der einen Seite ein Programm mit dem Assembler von DEBUG (mit Hilfe des Befehls "A" oder "a") in der Assemblersprache anhand eines Entwurfs eingegeben, so läßt sich der Programmablauf anhand des Entwurfs und der Wiedergabe auf dem Bildschirm leicht verfolgen; denn die Wiedergabe dieses Programms erscheint auf dem Bildschirm mit Hilfe des Befehls "U" (Unassemble, Rückassemblieren) in derselben Form wie der Entwurf - mit der kleinen Ausnahme, daß in dieser rückassemblierten Form der Maschinencode der Befehle angegeben wird; und jeder einzelne Befehl erscheint bei irgendwelchen Unterbrechungen des Programms in eben derselben Form wie auf dem Entwurf.

Dazu ein sehr simples Programmbeispiel: der Computer soll durch Abfragen feststellen, welche Zahl (von 00 bis FF, Hexadezimalzahlen) in den Speicherplatz mit Namen AL eingegeben wurde:

a) Entwurf (auf dem Papier):



b) Eingabe unter DEBUG: Bei der Systemanfrage von DOS ("A:" oder ähnlich) wird mit der Eingabe DEBUG (+ Zeilenschaltung = Enter) dieses Hilfsprogramm aufgerufen; bei der Systemanfrage von DEBUG (meist ein Bindestrich –) wird ein A (+Enter) eingegeben; das System gibt dann z.B. die hexadezimalen Zahlen 100A:0100 an, wobei die vor dem Doppelpunkt stehende Zahl anders lauten kann. Am Ende jeder Zeile eine Zeilenschaltung!

```
A: DEBUG
-A
100A:0100    MOV BL,0     (Die Befehls-Eingabe kann überall
100A:0102    CMP BL,AL    auch in Kleinbuchstaben erfolgen!)
100A:0104    JZ  10A
100A:0106    INC BL
100A:0108    JMP 102
100A:010A    INT 3
100A:010B    ^C           (mit CTRL-Break wird die
                          Eingabe abgebrochen)
```

Die beiden (Hexadezimal-) Zahlen am Beginn geben die Nummer des Speicherplatzes an, in dem der Befehl (in Maschinensprache!) beginnt; diese Zahlen liefert das System selbst, der Programmierer muß nur die rechtsstehenden Zeichen, die Befehle in Assemblersprache, eingeben; die Befehle in Maschinensprache sind verschieden lang; z.B. belegt der Maschinenbefehl für "INT 3" nur einen Speicherplatz. Die Zahlen bei den Sprüngen (JZ und JMP) geben an, zu welchem Speicherplatz - und dem darin enthaltenen Befehl -, d.h. zu welcher "Adresse" der Sprung unter Umständen erfolgen soll.

c) Wiedergabe des so eingeschriebenen Programms unter DEBUG mit Hilfe des Befehls U (unassemble). Die erste Zeile wird eingegeben (Rückassemblieren von 100 bis 10A), die restlichen gibt das System an:

```
-U 100 10A
100A:0100   B3 00      MOV    BL,00
100A:0102   38 C3      CMP    BL,AL
100A:0104   74 04      JZ     010A
100A:0106   FE C3      INC    BL
100A:0108   EB F8      JMP    0102
100A:010A   CC         INT    3
```

Gegenüber der Eingabe ist nur die Änderung erfolgt, daß der Maschinencode der Befehle mitangegeben ist. Hier sieht man, daß der Maschinencode für "INT 3" nur ein Byte (= eine zweistellige Hexadezimalzahl) verwendet, während die übrigen hier verwendeten Befehle alle ("zufällig") zwei Byte (also ein "Doppelbyte" oder "Word") brauchen.

Eine Schwierigkeit zeigt sich bei der Verwendung des DEBUG-Assemblers immer wieder: Man hat das Programm eingegeben, ausprobiert, freut sich, daß es läuft - und vergißt darüber, das Programm in Maschinencode mit Hilfe von DEBUG (mit Hilfe des Befehls W und Angabe der Länge des Programms in BX,CX!) abzuspeichern: Man kann die Arbeit (fast) von vorne beginnen!

Der große Nachteil dieser Art zu programmieren zeigt sich aber, wenn irgendeine Korrektur gemacht werden muß: im allgemeinen muß das Programm von der Stelle der Korrektur an neu eingegeben werden, und - was schwerwiegender ist - viele Adressenangaben für Sprünge, Unterprogramme, Daten usw. werden falsch und unbrauchbar; außerdem ist es schwierig nachzuprüfen, welche der derartigen Angaben durch die Änderung betroffen sind.

So soll in unserem Beispiel nun nachträglich festgelegt werden, daß nur gerade Zahlen vorkommen dürfen; dann könnte natürlich BL in der vierten Zeile unseres Programms jeweils um 2 vermehrt werden, die Abfrage der ungeraden

Zahlen ist ja überflüssig. Wir könnten, um dies zu erreichen, noch einen weiteren Befehl INC BL mit dem DEBUG-Befehl A 108 einfügen, dann aber würde der hier eingeschriebene Befehl JMP 0102 überschrieben, und wir müssen den Rest des Programms neu eingeben; also etwa so:

```
-A 108
100A:0108     INC  BL
100A:010A     JMP  102
100A:010C     INT  3
100A:010D     ^C
```

Wir brechen die Eingabe mit CTRL-Break ab und lassen uns das eingeschriebene Programm mit dem DEBUG-Befehl U 100 10C rückassemblieren. Das ergibt:

```
-U 100 10C
100A:0100     B3 00     MOV     BL,00
100A:0102     38 C3     CMP     BL,AL
100A:0104     74 04     JZ      010A
100A:0106     FE C3     INC     BL
100A:0108     FE C3     INC     BL
100A:010A     EB F6     JMP     0102
100A:010C     CC        INT     3
```

Und wenn wir dieses Programm laufen lassen, dann bleibt unser Computer "hängen": Wenn im Vergleich CMP BL,AL festgestellt wird, daß beide Zahlen gleich sind, dann verzweigt der nächste Befehl ("Springe bei Gleichheit") zur Adresse 010A, also zum Befehl JMP 0102, führt diesen Befehl aus, d.h. der Vergleich CMP BL,AL wird wieder ausgeführt und, weil sich bei diesen beiden Speicherplätzen nichts geändert hat, verzweigt der nächste Befehl zu JMP 0102 usw.usw. Wir müssen nach der Einfügung des zweiten INC BL also suchen, wo sich etwa dadurch eine Sprungweite geändert hat. In unserem Beispiel ist dies noch leicht: wir müssen mit dem DEBUG-Befehl A 104 den Sprungbefehl ändern:

```
-A 104
100A:0104     JZ   10C
100A:0106     ^C
```

Ein "Tüftler" wird nun vielleicht sagen, es gibt doch den Befehl ADD BL,2; wenn ich diesen anstelle des ersten INC BL eingebe, dann benutze ich auch nur eine Zeile, und das übrige Programm kann gleich bleiben. Es ist richtig, daß dieser Befehl (in Assemblersprache) nur eine Zeile beansprucht, er braucht aber in Maschinensprache **drei** Byte, und die Folgerungen sind ähnlich wie bei unserer ersten Änderung.

Verwendet man auf der anderen Seite einen sogenannten Makro-Assembler, dann muß der Programmierer neben der Kenntnis der Befehle auch noch über den verhältnismäßig umfangreichen Katalog der Regeln für die richtige Benutzung des Makro-Assemblers verfügen können. Schwerwiegender ist aber für den "Einsteiger", daß sich Entwurf und Eingabe so sehr von der rückassemblierten Wiedergabe des Programms auf dem Bildschirm unterscheiden, daß ein Verfolgen des Programmablaufs mit DEBUG einerseits und Entwurf/Eingabe andererseits nur nach hinreichend langer Übung gelingt. Damit ist der Vorteil des Makro-Assemblers entwertet, der an sich darin liegt, daß sich der Programmierer wenig um Sprungweiten und Adressen von Daten, Unterprogrammen usw. kümmern muß.

3. Die hier vorgeschlagene Lösung

Der vorliegende Assembler versucht, die Vorteile des DEBUG- und des Makro-Assemblers für den "Einsteiger" zu verbinden: die "Eingabe" eines Programms erfolgt in einer Form, die der rückassemblierten Form sehr nahe ist; durch eine einfache Verschlüsselung von Sprungweiten und (Daten-)Adressen wird das Programmieren und vor allem das Korrigieren von Programmen sehr erleichtert.

Die "Eingabe" erfolgt mit Hilfe des Text-Programms EDLIN, das in DOS neben DEBUG enthalten ist. EDLIN gestattet die zeilenmäßige Eingabe von "Texten", die Einfügung und das Löschen von Zeilen, in höheren Versionen auch das (blockweise) Ver-

schieben und Duplizieren von Zeilen - insgesamt also gerade das alles, was zum Schreiben eines maschinennahen Programms benötigt wird; im übrigen sorgt das Programm EDLIN selbständig für die Numerierung der Zeilen.

Beim Aufruf dieses Hilfsprogramms bei Systemanfrage mit dem Befehl EDLIN muß noch ein Dateiname angegeben werden, unter dem das später zu übersetzende (zu assemblierende) Programm auf der Diskette gespeichert wird; in dieser Form heißt das Programm allgemein "Quellenprogramm".

Das Quellenprogramm für unser obiges Beispiel wird also folgendermaßen eingegeben:

A: EDLIN PROG1.ED			PROG1.ED sei der Name unseres Quellenprogramms
*I			* ist die Systemanfrage von EDLIN I = ein Text soll zeilenweise eingegeben werden.
1:	MOV	BL,0	Zahl in al "erraten" (Dieser Zusatz wird auch eingegeben!)
2:0001	CMP	BL,AL	
3:	JZ	/2/	
4:	INC	BL	
5:	JMP	/1/	
6:0002	INT	3	
7: ^C			(Die Eingabe wird mit CRTL-Break abgebrochen)
*E			Durch den Befehl E wird PROG1.ED abgespeichert und EDLIN beendet.

Wie schon gesagt, erfolgt die Numerierung der Zeilen automatisch durch das Programm EDLIN. Die übrigen Zeichen müssen eingegeben werden.

Ein solches Quellenprogramm wird durch den Assembler übersetzt und auf Wunsch als neues Programm unter einem (andern) Namen, z.B. PROG1.PRO, abgespeichert. Dieses Programm in Maschinensprache, in dem die einzelnen Befehle nur noch im Maschinencode eingeschrieben sind, heißt "Objektprogramm".

Vergleichen wir nun ein Quellenprogramm (das Quellenprogramm unseres Beispiels) mit dem entsprechenden Objektprogramm, das wir mit Hilfe von DEBUG rückassemblieren:

Quellenprogramm:

1	2	3	4	5
1:		MOV	BL,0	Zahl in al "erraten"
2:	0001	CMP	BL,AL	
3:		JZ	/2/	
4:		INC	BL	
5:		JMP	/1/	
6:	0002	INT	3	

Objektprogramm (rückassembliert):

1	2	3	4
100A:0100	B3 00	MOV	BL,00
100A:0102	38 C3	CMP	BL,AL
100A:0104	74 04	JZ	010A
100A:0106	FE C3	INC	BL
100A:0108	EB F8	JMP	0102
100A:010A	CC	INT	3

Das eigentliche Objektprogramm - das Programm im Maschinencode - ist nur entsprechend Spalte 2 byteweise hintereinander im Speicher geladen, hat also im Speicher die Form

B30038C37404FEC3EBF8CC;

die übersichtliche Anordnung auf dem Bildschirm und die übrigen Spalten des rückassemblierten Objektprogramms liefert der Befehl U.

Man sieht die Übereinstimmung der beiden Programmdarstellungen in der jeweils dritten und vierten Spalte; nur zwei kleine Unterschiede sind zu bemerken:

- die Null in der ersten Zeile erscheint im (rückassemblierten) **Objektprogramm** zweistellig - allgemein: Zahlen erscheinen zwei- oder vierstellig, eventuell mit vorlaufenden Nullen;
- die Sprungadressen sind anders gekennzeichnet: Im (rückassemblierten) **Objektprogramm** ist in der Spalte 4 die Adresse angegeben, wohin der Sprung (unter Umständen) erfolgen soll - diese Adresse ist in Spalte 1 (hinter dem Doppelpunkt) zu finden; beim **Quellenprogramm** hingegen signalisieren die

Schrägstriche //, daß der Sprung zu einer bestimmten "Marke" erfolgen soll, die in Spalte 2 des Quellenprogramms angeführt ist. Diese Spalte wird nicht mitübersetzt, dient aber dem Assembler für die Berechnung der Adressen in Spalte 4 des Objektprogramms.

Die Spalte 5 des Quellenprogramms ist hier ein "Kommentar", der kurz angibt, was dieses Programm leisten soll. Bei der Übersetzung wird dieser Kommentar offensichtlich nicht beachtet.

Wenn wir nun (wie oben) annehmen, daß nur gerade Zahlen zu "erraten" sind, können wir mit Hilfe von EDLIN die vierte Zeile des Quellenprogramms in ADD BL,2 (addiere 2 zu BL) ändern, oder eine neue fünfte Zeile - ein zweites INC BL - einschieben, die Sprungadressen in Zeile 3 und 5 (bzw. nach einem Einschub Zeile 6) ändern sich genausowenig wie die Marken in Zeile 2 und 6 (bzw. 7). Die Übersetzung durch den Assembler liefert wieder ein funktionierendes Programm. Die Änderung (Korrektur) war aber sehr einfach, ohne daß viel Sorgfalt um richtige Adressierung aufgewendet werden mußte.

Da Korrekturen nur im Quellenprogramm vorgenommen werden, das mit der Beendigung von EDLIN abgespeichert wird, ist dieses immer "aktuell" und jederzeit als Datei verfügbar, die "Eingabe" ist zugleich "Protokoll" des Programms.

Natürlich müssen beim Eingeben der einzelnen Befehle mit Hilfe von EDLIN einige Regeln beachtet werden: der Assembler kann in keiner Weise "mitdenken" und bei Ungenauigkeiten aus den Umständen gutwillig interpretieren, was gemeint sei (wie oft würde man sich das beim Programmieren wünschen!). Jedes eingegebene Zeichen ist zugleich eine Anweisung für den Assembler, einen bestimmten Übersetzungsvorgang einzuleiten, abzubrechen, in einer bestimmten Weise fortzusetzen usw. Und dies alles muß dem Assembler in eindeutiger Weise mitgeteilt werden, d.h. eben, daß beim Eingeben bestimmte Regeln zu beachten sind. Diese Regeln sollen nun angegeben werden.

II. Die Erstellung des Quellenprogramms

Das Quellenprogramm wird mit Hilfe von EDLIN eingegeben. Es darf nicht mehr als 65.000 Zeichen umfassen.

Jede Zeile des Quellenprogramms hat folgende Struktur:

xxxxx: yyyy (Tab) BEF (Tab,ZS) OPERAND (ZS, Tab + Kommentar)

Zu den einzelnen Symbolen bzw. Abkürzungen:

xxxxx: Diese **Dezimalzahl** mit Doppelpunkt wird als Zeilenangabe vom EDLIN-Programm nur auf dem Bildschirm (und eventuell dem Drucker) als Hilfe für den Programmierer ausgegeben, aber nicht abgespeichert.
Die Eingabe von seiten des Programmierers beginnt **immer hinter** dieser Zeilenangabe.

yyyy eine höchstens vierstellige **Dezimalzahl** als Marke für einen Befehl (z.B. Beginn eines Unterprogramms), die natürlich nur dort eingefügt wird, wo ein Befehl markiert werden soll.
Wichtig: Jede Marke darf in einem Programm nur einmal verwendet werden. (Es dürfen aber beliebig viele Adressen in Operanden - s.u. - auf ein und dieselbe Marke zielen!)
Empfehlenswert: Alle Marken sollten vierstellig, also jeweils mit vorlaufenden Nullen (z.B. 0004, 0025) geschrieben werden; dies erleichtert die Suche dieser Marken (mit Hilfe des Suchbefehls von EDLIN) bei der Korrektur: es gibt (wahrscheinlich) mehrere Marken die mit 4 (und einem

nachfolgenden TAB = Tabulator) enden, aber nur **eine** 0004 (mit nachfolgendem TAB).

(Tab) Hinter der Marke, bzw. am Anfang der Zeile (wenn keine Marke eingefügt wird) steht ein **Tabulator.** Dieser (erste) Tabulator in einer Zeile (beendet die Marke und) signalisiert, daß mit dem folgenden Zeichen das Befehlswort beginnt.
Beachte: Am Zeilenanfang bzw. zwischen Marke und Befehlswort müssen Leerzeichen (Blanks) vermieden werden!

BEF Nach dem (ersten) Tabulator muß das "mnemotechnische" oder "mnemomonische" Befehlswort eingegeben werden; "mnemotechnisch" heißt, daß dieses Befehlswort das Gleiche bedeutet wie der entsprechende Maschinencode, aber eben leichter zu "memorieren", im Gedächtnis zu behalten (und leichter zu verstehen) ist, als die wenig besagenden Zahlen. Dieses wird wiederum mit einem **Tabulator** abgeschlossen.

(Tab,ZS) Dieser zweite Tabulator schließt das Befehlswort (im mnemotechnischen Code geschrieben) ab und signalisiert dem Assembler, daß das folgende Zeichen zum sogenannten Operanden (siehe unten) gehört.
Wenn das Befehlswort prinzipiell ohne Operanden steht (z.B. LODSB, CBW; Vorsicht bei RET!), bedeutet hier ein Tabulator dasselbe wie ansonsten der **dritte** Tabulator hinter dem Operanden (siehe unten); diese Befehlswörter werden normalerweise hier mit einer Zeilenschaltung (ZS) abgeschlossen.
Beachte: Zwischen BEF und OPERAND müssen Leerzeichen (Blanks) vermieden werden!

OPERAND Nach dem (zweiten) Tabulator folgt der Operand, der festlegt, auf welchen Speicher bzw. welche(s) Register (s.S. 20) der Befehl wirken soll, bzw. bei "Sprungbefehlen", wohin dieser Sprung führen soll.

Grundsätzlich gilt, wenn **zwei** Speicher/Register angegeben sind, daß der Befehl **vom rechts** stehenden **zum links** stehenden wirkt und das Ergebnis im linken steht (sofern sich eine Veränderung des links stehenden Speichers/Registers ergibt); das rechts stehende bleibt **immer** unverändert.

Die Behandlung von Konstantzahlen innerhalb des Operanden:

Der Assembler interpretiert alle Zahlen als Hexadezimalzahlen und behandelt sie von selbst richtig als zwei- oder als vierstellige Hex-Zahlen (z.B. die Hex-Zahl F als 0F oder 000F). **Ausnahme:** die zwischen zwei Schrägstrichen /../ stehenden Zahlen werden nur als **Dezimal**zahlen verstanden und behandelt (die Hex-Ziffern A,B,C,D,E, und F sind hier fehl am Platze!) und als "Adresse", d.h. als Verweis auf die entsprechenden Marken an anderer Stelle des Programms bezogen.

JMP /273/ heißt: Sprung zum Befehl, der mit 0273 markiert ist.

Zu derartigen Dezimaladressen kann eine (bis vierstellige) Hexadezimalzahl **addiert** werden, z.B. /24/+F. So kann mit der Adresse /45/+3 das vierte Byte in der Datenzeile, die mit 0045 markiert ist, erreicht werden, so daß also unter Umständen nicht jedes einzelne Byte markiert werden muß, was auch gar nicht möglich wäre (s.u. im Abschnitt "Eingabe von Festzahlen" und "Sprungbefehle, Verzweigungen, Unterprogrammaufrufe"). Welche Vorteile dies bietet, können wir erst anhand von Beispielen

einsehen (s.u. Abschnitt IV,2).

Beachte: Im Bereich des Operanden müssen Leerstellen (Blanks) - außer nach BY bzw. WO, siehe S. 20 - vermieden werden!

(ZS,Tab + Kommentar) Der Operand - und damit die Befehlszeile - wird normalerweise mit einer Zeilenschaltung (ZS) abgeschlossen. Jedoch kann jede Befehlszeile (außer diejenige, in der RET ohne Zusatz steht) anstatt mit der ZS mit einem Tabulator abgeschlossen und mit einem Kommentar versehen werden. Bei der Übersetzung werden alle Zeichen, die ab diesem Tabulator noch in der Zeile stehen, übergangen.

Der Kommentar darf nicht so lang sein, daß die (mit EDLIN eingebbare) Zeile überschritten wird; jeder Kommentar kann aber in der nächsten Zeile (nach dem abschließenden Tabulator) fortgesetzt werden.

So könnten die ersten beiden Zeilen unseres Beispiel-Quellenprogramms auch lauten:

1:	MOV	BL,0	Zahl in AL "erraten"; ein sehr
2:0001	CMP	BL,AL	simples Programm.

Die wohl umfangreichste Aufgabe für den zukünftigen Programmierer liegt - und dies bei allen Programmiersprachen - darin, **die Befehle** kennenzulernen; zu den Befehlen der Assemblersprache müssen zunächst einige allgemeine Bemerkungen gemacht werden. Es gibt im Assembler "eigentliche" und "uneigentliche" Befehle. Wir beginnen mit den **"uneigentlichen" Befehlen.**

1. Uneigentliche Befehle

In der Assemblersprache kommen zwei Befehlswörter vor, die nur Anweisungen an den Assembler sind, nicht aber als Befehle weiter übersetzt werden:

REM und
DB bzw. DW

REM (Abkürzung von **Rem**ark)
Wenn der Assembler in einer Zeile das "Befehlswort" REM (dahinter TAB) findet, dann übergeht er bei der Übersetzung diese ganze Zeile. Mit dem "Befehl" REM können also beliebig lange Kommentare an jeder beliebigen Stelle in das Programm eingefügt werden.

Ausnahme: Wenn der "Befehl" REM in der **ersten** Zeile des Quellenprogramms steht, dann nimmt der Assembler die Zeichenfolge, die den "OPERANDEN" darstellt, als den Namen, unter dem das **übersetzte** Programm **automatisch** auf die Diskette abgespeichert wird (sofern die Zeichenfolge ein zulässiger Dateiname ist),

z.B.

```
REM     PROG1.PRO
```

Stünde dieser "Befehl" in der ersten Zeile unseres Beispiels (er könnte auch nachträglich "eingeschoben" werden), dann würde der Assembler automatisch nach dem Übersetzen das Objektprogramm unter diesem Namen abspeichern (auf die Diskette und in deren Dateiverzeichnis, wo auch das Quellenprogramm abgespeichert ist; s. S. 40).

Steht in der ersten Zeile kein REM, dann wird das übersetzte Programm unter dem Namen des Quellenprogramms, jedoch mit der Erweiterung ".OBJ" abgespeichert.

Bei jedem neuen Übersetzen des Quellenprogramms wird beim erneuten Abspeichern des Objektprogramms unter dem gleichen

Namen das bereits abgespeicherte Objektprogramm überschrieben und geht verloren. Das **Quellen**programm bleibt in der jeweils unmittelbar vorausgehenden Fassung als EDLIN-Datei unter dem gleichen Namen, jedoch mit der Erweiterung ".BAK" (in unserem Fall also "PROG1.BAK") erhalten.

DB bzw. **DW** (Abkürzung von **D**ata-**B**ytewise bzw. **D**ata-**W**ordwise, **D**aten-**B**yteweise bzw. **D**aten-**W**ortweise)

Dieser uneigentliche Befehl dient dazu, Daten für das Programm bereitzustellen bzw. Speicherplätze für Daten zu reservieren. Dieser Befehl kann prinzipiell an jeder beliebigen Stelle im Programm stehen, nur muß dafür gesorgt sein, daß die dadurch festgelegten Datenspeicherplätze im Programm**ablauf** immer übersprungen werden.

Der "Befehl" DB bzw. DW bedeutet für den Assembler die Anweisung, alle Zeichen (mit drei Ausnahmen, s.u.) des "OPERANDEN" als "Daten" einzeln hintereinander abzulegen; sollten diese "Daten" ein Text sein, so muß dieser in Anführungszeichen ".." oder '..' eingeschlossen sein; die übrigen durch Leerstellen (Blanks) bzw. Kommata getrennten Zeichen(folgen) werden im Falle von DB als zweistellige, im Falle von DW als vierstellige Hexadezimalzahlen interpretiert.

```
1:    DB     0 8f, 7C 8
2:    DW     0 37e 8
```

Die erste Zeile wird 00 8F 7C 08 abgespeichert, die zweite **00 00 7E 03 08 00** (die Zwischenräume zwischen den einzelnen Byte sind hier nur der Deutlichkeit halber eingefügt). Dabei fällt auf, daß bei der Wortabspeicherung (z.B. von 037E) die beiden Byte vertauscht **erscheinen** (7E 03); dies gilt ganz allgemein: das höherwertige Byte eines Wortes, das in der uns gewohnten Darstellung links vom niederwertigen steht (037E), wird in den höheren Speicherplatz abgelegt, steht also in der byteweisen Aneinanderreihung der Speicherplätze "rechts" oder "hinter" dem niederwertigen (7E 03). Auf diesen Umstand wird noch öfters hinzuweisen sein.

Sollte ein Text länger als eine EDLIN-Zeile sein, so kann er in einer neuen DB-Befehlszeile als OPERAND fortgesetzt werden.

Folgende DB-Befehlszeilen sind also gleichbedeutend:

```
    1:0067   DB   "Dieser Text wird abgespeichert"
und:
    1:0067   DB   "Dieser Text"
    2:       DB   " wird abgespeichert"
```

Ausnahmen:

/ Dieses Zeichen bedeutet die Anweisung an den Assembler, die folgenden **Dezimal**zahlen (bis zum zweiten "/") als Adresse zu interpretieren und die Speicherplatznummer der mit dieser Zahl markierten übersetzten Befehlszeile bzw. Datenzeile als Doppelbyte (Word) abzulegen.

Dieses Zeichen leitet eine genau definierte Zeichenfolge ein bzw. schließt diese ab: #2F;20# heißt für den Assembler, daß er 2F-mal das Byte 20 an dieser Stelle einfügt. Auf das erste Zeichen # folgt also eine höchstens zweistellige Hexadezimalzahl, darauf ein Strichpunkt (Semikolon) und wieder eine solche Zahl, darauf als Abschluß wieder das Zeichen #; die erste Zahl gibt an, wie oft die zweite als Byte abgelegt werden soll. Folgende beiden DB-Zeilen sind also gleichwertig:

```
DB     0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
DB     #C;F#
```

Hiermit kann also ein "Puffer" - eine Daten-"zeile", in die beim Programmablauf Daten, vor allem Texte, vorübergehend abgespeichert werden - in kurzer Weise definiert werden.

Tab Ausdrücklich sei darauf hingewiesen, daß auch in DB-/DW-Befehlszeilen ein Tabulator außerhalb von Anführungszeichen einen **Kommentar** einleitet. Von der Befehlszeile

0033 DW 0 (Tab) Ein 0-Wort ist unter Marke 33 reserviert

werden nur zwei 00-Bytes, eine vierstellige Hex-Zahl 0000, abgespeichert.

2. Eigentliche Befehle

Die einzelnen Befehle, die in Abschnitt VI alle angegeben sind, werden in der unassemblierten Form eingegeben. Allgemeine Bemerkungen können nur zu den Adressen in den Operanden der Befehle, zur Eingabe von Festzahlen und zum Operanden der Sprungbefehle gemacht werden.

a. Adressen der Befehle

In den Operanden der Befehle werden - abgesehen von den Sprungbefehlen - die Register benannt bzw. die Speicherplätze "adressiert", die durch den Befehl betroffen sind. Als "Register" werden die Speicherplätze innerhalb des Prozessors, des eigentlichen Rechenwerkes, bezeichnet; mit ihnen kann sehr schnell gearbeitet werden. Die Speicherplätze des Arbeitsspeichers werden kurz nur als "Speicherplätze" bezeichnet; der bearbeitende Zugriff auf diese Speicherplätze ist etwa zehnmal langsamer als bei den Registern. Die Register werden mit "Namen" bezeichnet (s. S. 21), die Speicherplätze sind nur durchnumeriert. - Bei einigen Befehlen kann im Operanden zusätzlich noch die Angabe erforderlich sein, ob es sich um eine Byte-Operation oder eine Word-Operation (zwei Byte hintereinander betreffend) handelt. Diese Angaben erscheinen rückassembliert je nach DOS-Auflage als "B," bzw. "W," oder "BY " bzw. "WO ", schließlich

"BYTE PTR " bzw. "WORD PTR " (die Leertaste vor dem abschließenden Anführungszeichen in den letzten vier Fällen ist unverzichtbar); der Assembler akzeptiert alle diese Formen.

Nur die in Abschnitt VI jeweils bei der Darstellung der Befehle angegebenen Kombinationen von Registern und Speicherplätzen sind möglich. Allgemein gilt:

In den Operanden kann **höchstens einmal** die Bezeichnung eines **Speicherplatzes** vorkommen, so daß z.B. die Übertragung eines Speicherplatzinhaltes in einen anderen Speicherplatz nicht unmittelbar erfolgen kann; das gleiche gilt für die **Segmentregister** (CS, DS, ES und SS).

Festzahlen können grundsätzlich nur rechts vom Komma stehen (außer sie werden als Adresse eines Speicherplatzes verwendet; dann aber stehen sie in eckigen Klammern; s.u.).

Die **Register** werden durch ihren Namen festgelegt:
AX, BX, CX, DX, BP, SP, SI, DI und IP; das Statusregister F und die **Segment**register CS, DS, ES und SS. Alle diese Register sind zwei Byte lang (ein Word oder 16 Bit).

Die ersten vier können noch geteilt werden, so daß in manchen Operationen entweder nur das niederwertige Byte (-L, Low) oder nur das höherwertige (-H, High) angesprochen werden kann:
AH, AL; BH, BL; CH, CL; DH, DL

Beispiele:

```
INC    AH       betrifft das höherwertige Byte von AX
INC    CL       betrifft das niederwertige Byte von CX
MOV    AX,SI    überträgt den Wert von SI nach AX
```

Darüber hinaus haben die einzelnen Register feststehende **Sonderaufgaben**, die bei den jeweiligen Befehlen angegeben sind; z.B. wird CX bzw. CL bei manchen Befehlen automatisch als Zählerregister verwendet.

Der Befehlszähler IP kann nur durch Sprungbefehle beeinflußt werden.

Besondere Vorsicht ist bei Manipulationen mit dem "Stapelzeiger" SP geboten, denn dieser zeigt jeweils entsprechend einem festgelegten Mechanismus auf den Befehl, auf den aus Unterprogrammen zurückgegangen werden muß (s.u. Abschnitt IV,1).

Weiter sind die Register bei manchen Befehlen fest (oder höchstens durch den Befehl SEG... veränderbar) mit bestimmten Segmentregistern verbunden; so z.B. bei Stringbefehlen SI mit DS, DI mit ES; Stringbefehle sind Befehle, die vor allem für die Bearbeitung von Zeichenketten (= String) verwendet werden.

Ein Register besonderer Art ist das **Statusregister** F: in ihm haben die einzelnen Bits unabhängig voneinander nur die Funktion eines **Anzeigers** bzw. eines Schalters; in F heißen die einzelnen Bits allgemein "Kennzeichen" oder "Flag" und sind je mit einem eigenen Namen versehen. Vgl. Tabelle S. 113.

Schalterfunktionen haben die Kennzeichen D (direction) und I (interrupt) und werden **nur** durch eigene Befehle beeinflußt:

D = 0 heißt bei Stringbefehlen aufwärtszählen (Normalzustand), (String = Zeichenkette)(nur der Befehl CLD macht D=0)
D = 1 heißt bei Stringbefehlen abwärtszählen (STD macht D=1)
I = 0 (sperrbare) Unterbrechungsanforderungen der Hardware werden **nicht** zugelassen, (CLI macht I=0)
I = 1 (sperrbare) Unterbrechungsanforderungen der Hardware werden zugelassen (STI macht I=1).

Die übrigen Kennzeichen hängen vom Ergebnis derjenigen vorausgehenden Operation ab, die und soweit sie Einfluß auf das Statusregister hatte:

A (auxiliary carry, Übertrag von Bit 3 zu Bit 4), C (carry, Übertrag aus dem höchsten [oder niedersten] Bit), O (overflow, Über-

lauf = [**entweder** Übertrag aus dem höchsten Bit **oder** Übertrag aus dem zweithöchsten Bit]), P (parity, gerade Anzahl der Einsen im niederwertigen Byte, also Bit 0 bis 7), Z (zero, Null), S (sign, negatives Vorzeichen bzw. höchstes Bit mit einer Eins besetzt): jeweils ja = 1, nein = 0; z.B. heißt Z = 1: das Ergebnis der letzten Operation, die einen Einfluß auf das Statusregister hatte, ist Null.

Die **Segmentregister** haben jeweils eine besondere Bedeutung:

CS legt das Befehlssegment (Code-Segment) fest, innerhalb dessen das Programm eingeschrieben oder eingelesen wird; auf den einzelnen Befehl zeigt das Register IP (Instruction-Pointer).

DS ist das Daten-Segment, ES ein Hilfssegment (Extra-Segment).

SS ist das Stapel-Segment (oder Stack-Segment), in dem die Adressen für den Rücksprung aus Unterprogrammen gespeichert werden und Werte zur vorübergehenden Speicherung abgelegt werden können. Der Stapelzeiger ist SP (Stack-Pointer).

Wenn man die Operanden der in Kap. VI aufgeführten Befehle betrachtet, könnte man zur Auffassung kommen, daß die **Speicherplätze** immer nur durch einen "Offset" - das ist ein fester Wert, der im Maschinencode zwei Byte groß ist, oder eines der Register BX, BP, SI oder DI (in verschiedenen Kombinationen) - festgelegt werden. Da alle diese Werte nur von 0000 bis FFFF reichen können, wäre es nur möglich, einen entsprechend kleinen Speicherbereich (immerhin 64 kB) zu adressieren. Um aber einen größeren Arbeitsspeicher nutzen zu können, werden diese Offsets **grundsätzlich** vom System (ohne daß dies in der Befehlsform des Programms aufscheinen würde) **mit dem Wert eines Segmentregisters** nach folgendem Mechanismus verbunden: das Segmentregister wird um eine Hexadezimalstelle nach links verschoben (die niedrigste Stelle gleich Null gesetzt, was insgesamt eine Multiplikation mit "sechzehn" bedeutet) und dann der Wert des Offsets dazu addiert; wenn also z.B. das Segmentregister DS gleich 14F3 (hexadezimal natürlich) ist und das Register SI gleich 2587, dann

zeigt DS:SI auf den (absoluten) Speicherplatz, der durch folgende Rechnung festgelegt ist:

DS	14F30
SI	2587
DS:SI	174B7

DS:SI zeigt also auf den 174B8-ten Speicherplatz des Arbeitsspeichers (der **erste** Speicherplatz hat die Nr. **0**, DS:SI=0:0 zeigt darauf; der **zweite** die Nr. **1** mit DS:SI=0:1 usw.). Durch diesen Mechanismus ist es möglich, mit den Segmentregistern einen "Block" (eben ein Segment) von Speicherplätzen festzulegen, innerhalb dessen man die einzelnen Speicherplätze mittels eines Offsets (von 0000 bis FFFF) bestimmt; insgesamt erreicht man dadurch (segmentweise) den Zugriff auf einen Speicher mit einem Megabyte Umfang. - Im oben angenommenen Fall DS = 14F3 kann man also, mit SI als Offset von 0000 bis FFFF, die Speicherplätze 14f30 bis 24F2F erreichen:

DS	14F30	DS	14F30
SI	0000	SI	FFFF
DS:SI	14F30	DS:SI	24F2F

Wenn, wie oben, der Wert von SI 2587 ist, dann zeigt SI auf den 2588-ten Speicherplatz **innerhalb des Teiles des Arbeitsspeichers,** des "Segmentes", **der durch DS festgelegt ist,** nicht aber einfach (absolut) auf den 2588-ten Speicherplatz des Arbeitsspeichers, sondern - unter der Voraussetzung DS=14F3 - auf den 174B8-ten Speicherplatz des Arbeitsspeichers.

Meist arbeitet ein Programmstück - für den Anfänger wohl das ganze Programm - mit festen Segmentregisterwerten, also auch mit festem DS, so daß in der Speicherplatzbezeichnung nur der Offset beachtet werden muß. Durch diesen "Normalfall" verführt, läßt man leicht außer acht, daß in einem Programm eines der Segmentregister verändert wurde und vertraut ein paar Befehle

später darauf, das System müßte doch wissen, wohin der Offset weisen sollte; nun, das System ist nie so freundlich "mitzudenken", so daß der "richtige" Offset wegen des "falschen" Segmentregisterwertes auf einen nicht gewünschten Speicherplatz zeigt, was beim Programmablauf meist zu einem "Absturz" des Systems führt.

Aus den Befehlen ist, wie gesagt, nicht ersichtlich, daß der Offset mit bestimmten Segmentregistern verbunden wird; darüber hinaus ist aber auch nicht ersichtlich, **mit welchem** Segmentregister der Offset in einem Befehl automatisch vom System kombiniert wird. Die hier zuständige Regel muß sich der Programmierer einprägen:

> Wenn in der Speicherplatzbezeichnung das Register BP genannt wird, dann bezieht sich der Befehl auf das Stapelsegment, die absolute Adresse wird also mit SS gebildet, sonst beziehen sich alle Bezeichnungen auf DS - falls nicht ausdrücklich mit dem Befehl SEG... (bzw. CS: usw.; s. Abschnitt VI bei SEG) eine andere Verbindung vorgeschrieben wird.

Folgende Speicherplatzbezeichnungen sind (in Assemblersprache) möglich:

[BX+SI]	**absolut:** DS:(BX+SI), d.h. zu dem links verschobenen Wert von DS wird die Summe von BX und SI dazugezählt, das Ergebnis bezeichnet den absoluten Speicherplatz; **innerhalb des Segments DS** zeigt BX+SI auf den Speicherplatz, mit dem die Operation vollzogen werden soll.
[BX+DI]	mit DS
[BP+SI]	mit SS
[BP+DI]	mit SS
[SI]	mit DS

[DI]	mit DS	
[klmn]	mit DS; k,..,n sind Hexadezimalziffern (0 bis F), klmn ist eine ein- bis vierstellige Hexadezimalzahl.	
[/opqr/]	mit DS; o,p,q,r sind Dezimalziffern (0 bis 9), opqr eine ein- bis vierstellige Dezimalzahl, eine gültige Marke, d.h. eine Marke, die im Programm tatsächlich vorkommt.	
[/opqr/+klmn]	mit DS; Verbindung der beiden vorhergehenden Bezeichnungen.	
[BX]	mit DS	
[BX+SI+klmn]	mit DS	k,l,m,n wie oben!
[BX+DI+klmn]	mit DS	**Statt klmn kann auch**
[BP+SI+klmn]	mit SS	**/opqr/ oder /opqr/+klmn**
[BP+DI+klmn]	mit SS	**stehen!**
[SI+klmn]	mit DS	
[DI+klmn]	mit DS	
[BP+klmn]	mit SS	
[BX+klmn]	mit DS	

Die Hexadezimalszahlen klmn bzw. die Zahlen /opqr/ oder die Summen /opqr/+klmn in der letzten Gruppe werden **"Displacements"** genannt. Dabei kann klmn nach /opqr/ auch ein **negatives** Vorzeichen haben.

Beim **Rück**assemblieren erscheinen alle diese Bezeichnungen in der gleichen Form; Ausnahmen gibt es natürlich bei den eingegebenen Festzahlen: die Zeichen / erscheinen nicht mehr; alle Zahlen erscheinen in hexadezimaler Darstellung, entweder in Byte- oder in Word-Größe, also zwei- oder vierstellig; die Summen /opqr/+klnm sind richtig berechnet und werden hexadezimal dargestellt; und **wichtig:** vierstellige Hexadezimalzahlen, deren neun höchste Bit besetzt sind (also alle Hex-Zahlen von +FF80 bis +FFFF) werden als negative Zahlen interpretiert

und deshalb mit negativem Vorzeichen aufgeführt, statt +FFFE erscheint also -0002.

Auf den wesentlichen Unterschied zwischen SI und [SI] (als Beispiel) soll hingewiesen werden:

MOV SI,AX	heißt: übertrage den Wert des Registers AX in das Register SI
MOV [SI],AX	hingegen: übertrage den Wert des Registers AX in den Speicherplatz DS:SI, also in den durch SI festgelegten Speicherplatz im DS-Segment (genauer: übertrage den Wert von AL in den durch DS:SI und den Wert von AH in den durch DS:(SI+1) festgelegten Speicherplatz).

b. Zur Eingabe von Festzahlen

Häufig ist es notwendig, durch das Programm bestimmte Zahlen in ein Register einzutragen oder beispielsweise die Zahl (ein Zeichen im hexadezimalen Code) in einem Register mit einer bestimmten festen Zahl (mit einem bestimmten Zeichen) zu vergleichen. Diese "Festzahlen" können wie die Displacements in der zweiten Gruppe der Speicheradressierung eingegeben werden:

klmn	Hexadezimalzahl, ein- bis vierstellig.
/opqr/	ein- bis vierstellige Dezimalzahl; vom Assembler wird der **ein-Word-lange** Hexadezimalwert eingesetzt, der den Speicherplatz kennzeichnet, in dem der mit dieser Zahl markierte Befehl steht.
/opqr/+klmn	die Kombination "Marke + hexadezimale Verschiebung"; das Ergebnis ist **immer ein Word lang.**

+FZ diese Eingabe einer ein- oder zweistelligen Hexadezimalzahl **mit Vorzeichen** ist bei manchen (Word-) Befehlen möglich und bedeutet folgendes: das durch FZ definierte Byte wird als solches in den Maschinencode des Befehls eingeschrieben, aber zum Word verlängert gerechnet, wobei das höchste Bit von FZ in alle Bit des erforderlichen höherwertigen Byte eingeschrieben werden; also bei Eingabe von +00 bis +7F wird 0000 bis 007F gerechnet, von 80 bis FF wird FF80 bis FFFF gerechnet; der kleine Vorteil dieser Eingabe: der Maschinencode ist ein Byte kürzer als bei der Eingabe der vierstelligen Hexadezimalzahl.

Bei manchen Befehlen (MOV AL,.. ; MOV AX,....; CMP AL,..; CMP AX,....; siehe die entprechenden Befehle in Abschnitt VI) ist es auch möglich, direkt ein **Zeichen** einzugeben bzw. bei CMP "abzufragen": statt des hexadezimalen Codes des Zeichens bzw. der Zeichen (siehe Code-Tabelle im Anhang) können diese selbst unter Anführungsstrichen angegeben werden (bei Byte-Operationen mit AL natürlich nur ein Zeichen, bei Word-Operationen mit AX zwei Zeichen). Folgende Zeilenpaare im Quellenprogramm sind also gleichwertig:

```
        MOV AL,41
        MOV AL,"A"
und     CMP AL,61
        CMP AL,"a"
oder    CMP AX,6241
        CMP AX,"Ab"
```

Hier ist wieder auf die (scheinbar) unterschiedliche Behandlung von höherwertigem und niederwertigem Byte in den Registern und im Arbeitsspeicher hinzuweisen: Wenn im Speicher eine Textstelle mit "Aber" beginnt, steht der Code für "b" hinter dem Code für "A", also 41 62; werden diese beiden Byte als ein Wort nach AX geladen (z.B. mit LODSW), steht das zweite als höherwertiges in AH,

also bei der Darstellung von AX links von 41, die Reihenfolge der Buchstaben scheint verdreht: "bA". Da bei Zeichenfolgen die "Wertigkeit" aber keine Rolle spielt, sondern nur die Reihenfolge der Zeichen wichtig ist, verwaltet der Assembler die Eingabe von Zeichenfolgen bei den Befehlen, wo dies möglich ist, in der "normal" erscheinenden textbezogenen Art.

Bezüglich des **Rück**assemblierens vgl. die Bemerkung oben.

Die unter Anführungsstrichen angegebenen Zeichen erscheinen im Objektprogramm im hexadezimalen Code, bei Word-Operationen in anscheinend vertauschter Reihenfolge.

c. Sprungbefehle, Verzweigungen, Unterprogrammaufrufe

Sprungbefehle sind Befehle, die den Prozessor veranlassen, das Programm nicht mit dem darauf folgenden Befehl, sondern an einer bestimmten anderen Stelle des Programms fortzusetzen; Verzweigungen sind besondere Sprungbefehle, bei denen es von gewissen Bedingungen abhängt, ob ein Sprung erfolgt oder das Programm mit dem folgenden Befehl weitergeführt wird; Unterprogrammaufrufe schließlich sind Sprungbefehle zu bestimmten Teilen des Programms, zu "Unterprogrammen" (UP, s. dazu Abschnitt IV,1,a), die ihrerseits schließlich einen Sprung zu dem Befehl veranlassen, der dem aufrufenden Befehl folgt. - Hier wird nur das gemeinsame Merkmal von Sprungbefehlen, Verzweigungen und Unterprogrammaufrufen behandelt, daß sie den Prozessor veranlassen, (unter Umständen) als nächstes nicht den nachfolgenden Befehl, sondern den an einer anderen Stelle des Programms - an einer anderen "Adresse" - stehenden Befehl auszuführen; soweit sind alle drei einfach "Sprungbefehle".

Diese verschiedenen "Sprungbefehle" bilden eine gewisse Besonderheit; beim **Rück**assemblieren wird nämlich bei diesen Befehlen immer die effektive "Adresse" ausgerechnet und somit angegeben, wohin der Sprung tatsächlich erfolgt. Beim **Eingeben** des

Programms kann im allgemeinen aber nur die **Sprungweite** oder die **dezimale Marke** des Ziels eingegeben werden, da ja die hexadezimale Ziel-"**Adresse**" des Objektprogramms vor der Übersetzung noch nicht feststeht.

Somit sind folgende Assembler-Operanden für Sprungbefehle vorgesehen:

a) Die **indirekte Angabe** des Ziel-Speicherplatzes geschieht mit Hilfe der Register oder des Inhalts festadressierter Speicherplätze; diese indirekte Angabe kommt nur bei JMP und CALL vor und wird bei diesen Befehlen (s. Abschnitt VI) besprochen.

b) Die **direkte Angabe** kann bei JMP und CALL durch hexadezimale Angabe von Segment und Zeiger erfolgen, z.B.

JMP A94:133

Durch diesen Befehl wird CS gleich 0A94 und IP gleich 0133; dies bedeutet den Sprung zu dem Befehl, der im absoluten Speicherplatz 0AA73 beginnt. **Rück**assembliert werden hier die Hex-Zahlen vierstellig (evtl. mit vorlaufenden Nullen) angegeben.

c) Die **direkte Angabe** kann weiter bei allen Sprungbefehlen durch die dezimale Adresse /opqr/ erfolgen; diese Adresse gibt an, zu welchem mit dieser Zahl **markierten** Befehl der Sprung vollzogen werden soll; der Assembler rechnet die Sprungweite aus.

Beispiel:

22:	JMP	/55/	Sprung zu Marke 0055
....			
7526:0055	INC	SI	Irgendeine Programmfortsetzung.

Diese direkte Angabe mit Marken hat den Vorteil, daß bei einer Korrektur des Quellenprogramms (z.B. durch Einschieben oder Löschen von Befehlszeilen) überhaupt keine Rücksicht auf Sprungbefehle gemacht werden muß, solange an den Marken selbst und an den auf sie zielenden Adressen nichts geändert wird. Ein Nachteil ist damit allerdings verbunden: Da man selbstverständlich eine Marke nur **einmal** verwenden darf (sonst "wüßte" der Assembler ja nicht, welche von beiden gleichlautenden Marken im einen oder andern Fall gilt), muß man Protokoll darüber führen, welche Marken schon verwendet wurden. (Am einfachsten geschieht dies dadurch, daß man sich eine Liste der Zahlen von 1 bis 100 und weiter zurechtlegt und jedesmal die entsprechende Zahl durchstreicht, wenn man sie als **Marke** verwendet hat).

Bei einiger Übung, wenn man kleine Passagen des Programms schon sicher überblickt, kann man sich nahe beisammenliegende Marken dadurch ersparen, daß man zu /opqr/ hexadezimal die Anzahl der Byte addieren läßt, die zusätzlich übersprungen werden müssen.

Beim obigen Beispiel:

22:	JMP	/55/ + 1	Sprung zum Befehl, der ein Byte
....			hinter Marke 0055 beginnt (der
....			Maschinencode von INC SI ist nur ein
....			Byte lang), hier also zu Zeile 7527
7526:0055	INC	SI	
7527:	ADD		Irgendeine Programmfortsetzung.

Bei leicht zu überblickenden Sprungweiten wird man einer der nächsten Möglichkeiten, Sprünge festzulegen, den Vorzug geben:

d) Die **Sprungweite** kann durch die **hexadezimal** angegebene **Anzahl der zu überspringenden Byte** festgelegt werden. Dabei

gilt, daß J... 0 (J... steht für irgendeinen Sprungbefehl; der Operand ist "Null") **fortfahren mit dem nächsten Befehl** bedeutet, die zu überspringenden Byte werden also von **diesem** Befehl aus gerechnet. Weiter wird bei den "kurzen" Sprüngen das höchstwertige Bit als Vorzeichen gedeutet: 7F bedeutet einen Sprung um (dezimal) 127 Byte vor, 80 einen Sprung um (dezimal) 128 Byte zurück; vgl. dazu die Umrechnungstabelle dezimal-hexadezimal im Anhang. Beim "langen" Sprung JMP bzw. bei CALL muß beachtet werden, daß ein eventueller "Übertrag" in eine fünfte Hexadezimalstelle verloren geht, so daß im allgemeinen der Operand FFFA gleichbedeutend ist mit "sechs Byte zurück!".

Beispiel:

12:	JC	3	bei C = 1 Sprung zum Befehl CALL.. in Zeile 14,
13:	JMP	FFE0	weil der Befehl JMP.. im Maschinencode drei
14:	CALL	139	Byte lang ist. Dieser JMP springt zu einem Befehl, der (vom Befehl CALL aus gerechnet) 32 Byte weiter vorne im Programm beginnt. Der letzte Befehl, CALL, ruft ein Unterprogramm auf, das (hexadezimal:) 139 Byte weiter hinten im Programm beginnt (s.u.!).

Beachte: Diese Form empfiehlt sich **nur**, wenn nur wenige Befehle oder Daten zu überspringen sind, deren **Byte-Länge im Maschinencode** bekannt ist.

Der Operand in Zeile 14 scheint also ziemlich unsinnig. Jedoch kann es in einem gut getesteten und öfters zu verwendenden Unterprogramm praktisch sein, einen ursprünglich vorhandenen Sprung zu einer Marke durch einen so definierten Sprung zu ersetzen (die Übersetzung des ursprünglichen Sprunges liefert im Objektprogramm die richtige Angabe dieser Sprungweite): in diesem UP kommt dann keine zusätzliche Marke (außer der Aufruf-Marke in der ersten Zeile des UP) mehr vor und damit wird die Gefahr einer Doppelbesetzung einer Marke beim Kopieren dieses UP in ein anderes Programm besser vermieden.

Rückassembliert wird hier die tatsächliche Adresse innerhalb des

(gleichen) CS-Segmentes erscheinen (im Beispiel: im Operanden der Zeile 12 die Adresse des CALL-Befehls).

e) Die **Sprungweite** kann weiter durch die **dezimale Differenz der Zeilenzahlen** (dezimal zweistellig - also unter Umständen mit vorlaufender Null - **mit Vorzeichen**) angegeben werden (hier zeigt sich ein Vorteil der Programmeingabe mit EDLIN!).

Beispiel:

34:	JZ	-08	bedingter Sprung zu Zeile 34-8 = 26
35:	JA	+32	bedingter Sprung zu Zeile 35+32 = 67

Hier liegt also eine Ausnahme gegenüber der allgemeinen Regel der Zahlendarstellungen vor: im Operanden von "Sprüngen" (Befehlen, die mit J beginnen, bei LOOP.. und bei CALL..) wird das Ziffernpaar **hinter einem Vorzeichen dezimal** interpretiert!

Rückassembliert wird auch hier die tatsächliche Adresse innerhalb des (gleichen) CS-Segmentes erscheinen.

Auf einen Nachteil dieser offensichtlich ganz einfach zu handhabenden Methode sei aber hingewiesen: Wenn bei einer Korrektur des Quellenprogramms **Zeilen** eingeschoben oder gelöscht werden müssen, sind alle derartig definierten Sprünge falsch, die über diese Korrekturstellen hinwegzeigen.

Bei der Eingabe von Sprüngen nach vorne zeigt sich bei dieser Angabe der Sprungweite ein Vorteil von EDLIN: Zunächst wird in der Zeile des Sprunges nur das Befehlswort angegeben - damit ist für diesen Sprungbefehl eine Zeile reserviert; nachdem im weiteren Verlauf der Programmeingabe die Zeile geschrieben ist, zu der gesprungen werden soll, wird die Eingabe abgebrochen, die unvollständige Zeile aufgerufen (durch Angabe der Zeilennummer mit Eingabetaste), ergänzt und nach der Eingabe mit

dem Befehl #I die Fortsetzung der Eingabe an der zuvor abgebrochenen Stelle aufgerufen.

Bei größerer Übung im Programmieren mit dem vorliegenden Assembler wird man diese letzte Methode der Sprungfestlegung, so weit wie möglich, der Angabe von Adressen (/.../) die auf Marken zielen, vorziehen; die Arbeit zu registrieren, ob man eine bestimmte Marke schon verwendet hat oder nicht, wird dadurch meist sehr eingeschränkt.

So könnte das Quellenprogramm im oben angeführten Beispiel auch lauten:

```
1:    MOV    BL,0
2:    CMP    BL,AL
3:    JZ     +03
4:    INC    bl
5:    JMPS   -03
6:    INT    3
```

III. Ablauf der Übersetzung und Fehlermeldungen

Nachdem das Quellen-Programm mit Hilfe von EDLIN eingegeben und mit dem EDLIN-Befehl E abgeschlossen und abgespeichert ist, muß die Assembler-Programmdiskette in das aktuelle physische Laufwerk - ein logisches Laufwerk ist ungeeignet - gegeben werden (natürlich kann zuvor, wenn vorhanden, ein anderes Laufwerk mit dem Befehl "A:" oder "B:" zum aktuellen Laufwerk gemacht werden, so daß die Diskette mit dem Quellenprogramm in der bisherigen Diskettenstation verbleibt).

Jedenfalls muß die Assembler-Programmdiskette im **aktuellen physischen Laufwerk** sein, damit das Assemblerprogramm nach der entsprechenden Systemanfrage mit dem Befehl

ASS

- ohne sonstigen Zusatz - aufgerufen werden kann.

Ist dies geschehen, verlangt das System nach einiger Rechenzeit, daß der **Name des Quellenprogramms** eingegeben wird. Diese Eingabe erfolgt in der Form

Laufwerkbuchstabe: [Pfad]Dateiname[.Erweiterung],

wobei die eingeklammerten Angaben, wenn sie überflüssig sind, weggelassen werden.

Das System akzeptiert **zunächst** jede nicht-leere EDLIN-Datei mit einem gültigen Namen als Quellenprogramm. Wenn der angegebene Name nicht gültig ist - also zu viele Buchstaben oder falsche Zeichen enthält oder nicht auf der Diskette vorhanden ist - werden entsprechende Fehlermeldungen angezeigt. Das System hält mit entprechender Fehleranzeige an, wenn die aufgerufene

Datei keine EDLIN-Datei bzw. nur eine leere (und damit uninteressante) Datei ist, und fordert zur Wiederholung der Prozedur auf.

Dann fragt das System an, ob eine **Marken-Adressen-Kontrolle** durchgeführt werden soll.

Diese empfiehlt sich dringend, wenn das Programm zum ersten Mal dem Übersetzungsvorgang unterworfen wird, oder wenn Korrekturen an Marken/Adressen stattgefunden haben.

Diese Kontrolle ist weiter eine Hilfe bei der Eingabe des Quellenprogramms, wenn nach einer Unterbrechung der Arbeit die Übersicht über die verwendeten Marken/Adressen vielleicht verloren gegangen ist. (Nach dieser Kontrolle kann der Assemblervorgang abgebrochen werden!)

Das System zeigt nach kurzer Rechenzeit an:
Den Abbruch des Assemblers, wenn eine Marke **zweimal** verwendet wurde; dies ist ein "fataler Irrtum", weil der Assembler nie "entscheiden" kann, welche der "beiden" Marken im Einzelfall einer Adresse zugeordnet werden soll.

Sonst gibt das System an:
1. Welche die größte verwendete Marke ist.
2. Welche Marken im Zahlenbereich von 1 bis zur größten Marke noch nicht verwendet wurden.
3. Welche Marken (bis jetzt) überflüssig sind, weil es keine Adresse gibt, die auf diese Marken zielt.
4. Welche Adressen sinnlos sind, weil sie auf Marken zielen, die nicht vorkommen.

Die Bemerkungen 2. und 3. sind nur Hilfen für eine weitere Quellenprogramm**eingabe**, nicht im eigentlichen Sinne Fehler, weil wegen dieser Mängel nichts falsch laufen kann, wenn das trotzdem übersetzte Programm verwendet wird.

Die Bemerkung 4. zwingt zur Korrektur des Quellenprogramms; jedoch kann eine vorläufige Übersetzung günstig sein - die sinnlose Adresse zeigt vielleicht auf ein fertig vorliegendes Quellenprogrammstück, das für die endgültige Übersetzung - z.B. als Unterprogramm - noch an das Quellenprogramm durch Kopieren angefügt werden soll.

Wenn die Marken-Adressen-Kontrolle zufriedenstellend verlaufen ist, kann man zur **Übersetzung** weitergehen (Drücken der Leertaste).

Bei der Übersetzung werden die Zeilennummern von Zeilen angezeigt, die am Beginn desjenigen Zeilenblockes stehen, der gerade bearbeitet wird; obwohl sich diese Zeilenzahlen sehr schnell ändern, wurde diese Anzeige gewählt, um die Tätigkeit des Assemblers augenfällig zu machen.

Wenn der Assembler einen **Fehler** entdeckt, bleibt er mit Anzeige der fehlerhaften Zeile (mit Zeilennummer) stehen und signalisiert durch einen senkrechten Strich, in welchem Bereich der Fehler liegt; es wird nur der "Bereich" angegeben, weil der Assembler im allgemeinen nicht feststellen kann, welcher Buchstabe falsch ist; z.B. kann das (falsche) Befehlswort ROT richtig RET oder ROL heißen, also das O oder auch das T falsch sein.

Der Fehler kann grundsätzlich sofort am Bildschirm **korrigiert** werden; die korrigierte Zeile wird neu übersetzt (und auf weitere Fehler untersucht). Die Korrektur wird **automatisch in das EDLIN-Quellenprogramm übertragen** und dieses am Ende der Übersetzung auf die Diskette anstelle des alten fehlerhaft geschriebenen Quellenprogramms abgespeichert.

Einige mögliche Fehler:

Fehler am **Beginn der Zeile:**
Die Zeile beginnt

- weder mit einer Dezimalziffer,
- noch mit einem Tabulator (vielleicht mit einem Leerschritt).

Fehler im **Bereich der Marke:**

a) Die verwendeten Zeichen sind keine Dezimalziffern.
b) Die verwendete Zahl ist größer als 9999.
c) Die Zahl wird nicht mit einem Tabulator abgeschlossen.

Fehler im **Bereich des Befehlswortes:**

a) Es wurde kein gültiges Befehlswort verwendet. Also z.B. bei LODS nicht angegeben, ob der Byte-Befehl LODSB oder der Word-Befehl LODSW gemeint ist.
b) Das Befehlswort, dem notwendig ein Operand folgt, wurde
   - nicht mit einem Tabulator abgeschlossen,
   - oder mit einer Zeilenschaltung abgeschlossen und der Operand fehlt.
c) Das Befehlswort, das nie einen Operanden hat, wurde
   - weder mit einem Tabulator
   - noch mit einer Zeilenschaltung abgeschlossen.

Fehler im **Bereich des Operanden:**
Hier sind die Fehlermöglichkeiten zu mannigfach, um sie alle aufzuzählen; im allgemeinen wird ein Vergleich des Geschriebenen mit den Befehlsdarstellungen in Abschnitt VI den Fehler entdecken lassen; schwerer zu entdeckende Fehler mögen folgende sein:

a) Überflüssige Leerzeichen sind eingefügt.
b) Byte- und Word-Operationen wurden im Operanden vermischt, so daß z.B. versucht wurde, das Byte AL in das (Word-) Register BX zu übertragen, oder eine Adresse /.../ in CL einzuschreiben.
c) Bei der Angabe von Festzahlen wurden die den Operanden einleitenden Zeichenfolgen "W," bzw. "L," (bzw. "BY " oder "WO ") vergessen.
d) Es wurde eine unerlaubte Manipulation mit dem Segmentregister CS versucht.

e) Der Operand wurde nicht mit einem Tabulator oder einer Zeilenschaltung abgeschlossen.
f) Klammern wurden vergessen.
g) Strichpunkt statt Komma u.ä.

Sprungweitenfehler

Es ist - besonders nach Korrekturen - möglich, daß ein (bedingter) "kurzer" Sprung durch Einschub von Zeilen oder langen Befehlen die grundsätzliche Reichweite eines kurzen Sprunges (dezimal -128 bzw. +127 **Byte**) überschreitet ("kurze" Sprünge über 60 **Zeilen** können mitunter gelingen!): In diesem Fall ersetzt der Assembler selbst den (bedingten) kurzen Sprung durch eine Befehls**folge**, die einen großen Sprung erlaubt, und ersetzt in allen möglichen Sprüngen in der Umgebung entsprechend die **Sprungweite, falls** diese durch die zu überspringende **Zeilenzahl** angegeben wurde (durch direkte oder indirekte Adressierung gekennzeichnete Sprünge werden nicht tangiert!). Sprünge, die durch die hexadezimale Angabe der Anzahl der zu überspringenden Byte festgelegt sind, werden nicht korrigiert!
Wenn allerdings bei der Korrektur des Quellenprogramms durch den Programmierer z.B. nach dem Befehl

```
23:    JNC    +06,
```

der also unter bestimmten Bedingungen zu Zeile 29 springen soll, ein Befehl eingefügt wird, so daß dieser Befehl in Zeile 23 nun zu Zeile 30 springen soll, dann muß die Zeile 23 mitkorrigiert werden:

```
23:    JNC    +07
```

Es ist also sehr ratsam, bei nachträglichen Einschüben oder Löschungen von Zeilen nachzusehen, ob damit nicht Sprungweiten von Befehlen der Umgebung betroffen werden.

Nach der Übersetzung

NB.: Die Zeilenzahl, die nach der Übersetzung stehen bleibt, ist nicht unbedingt gleich der letzten Zeilenzahl des EDLIN-Quellenprogramms. Da nämlich Sprungbefehle "nach vorne" bzw. Adressen von Programmteilen, die zum Ende des Programms hin liegen, nicht berechnet werden können, so lange das Programm nicht vollständig übersetzt vorliegt, werden hier beim Assemblieren zunächst Scheinadressen bzw. Scheinsprungweiten eingesetzt, die entsprechenden Zeilen vorgemerkt und, wenn schließlich auch deren Ziele festliegen, der Reihe nach noch einmal übersetzt - so bleibt natürlich als letzte die Zeilenzahl stehen, in der zuletzt ein solcher nachträglich zu berechnender Befehl vorkam.

Wenn in der ersten Zeile des Quellenprogramms ein REM-Befehl mit Namensangabe stand, wird das übersetzte Programm unter diesem Namen, sonst unter dem Namen des Quellenprogramms, jedoch mit der Erweiterung ".OBJ" abgespeichert.

Schließlich fragt das System an, ob das übersetzte Programm sofort unter **DEBUG** laufen soll. Wird dies gewünscht, dann wird DEBUG mit dem neuen Objektprogramm geladen, so daß dieses sofort mit allen DEBUG-Unterstützungen laufen kann, nachdem die DEBUG-Systemanfrage (ein Bindestrich) erschienen ist.

Diese DEBUG-Unterstützung ist nur möglich, wenn auf der Assembler-Programmdiskette die Dateien COMMAND.COM und DEBUG.COM des verwendeten Systems vorhanden sind. Diese beiden Dateien müssen also vor der Arbeit mit dem Assembler auf die Programmdiskette kopiert werden.

IV. Allgemeine Überlegungen zum Programmieren

1. Der Umgang mit dem Stapel

Unter DEBUG kann mit dem Befehl R (Register) der aktuelle Zustand aller Register abgefragt werden. Unmittelbar nach dem Laden von DEBUG zeigt dieser Befehl, daß alle Segmentregister auf den gleichen Wert und alle Register gleich Null gesetzt sind, außer IP und SP.
Der Instruction Pointer, der Befehlszeiger, steht auf 0100, dem ersten Speicherplatz, den DEBUG für die Eintragung eines Befehls zur Verfügung stellt und zugleich dem allgemeinen Startpunkt (innerhalb des bereitgestellten CS) eines jeden Programms, ob es nun unter DEBUG läuft oder nicht: jedes Programm wird von DOS von CS:0100 an geladen und jeweils an diesem Platz beginnt der Start-Befehl. Nur unter dieser Voraussetzung ist es möglich, in einem Programm feste Adressen (für Daten oder für Sprünge) anzugeben: eine Entnahme von Daten im Objektprogramm aus einem bestimmten Speicherplatz (z.B. 03F7) kann nur dann "programmgemäß" verlaufen, wenn diese Daten beim Laden des Programms von DOS auch immer an diese Stelle (innerhalb von CS) abgelegt werden; würde DOS ein Programm einmal von CS:0000 ab, das andere Mal von CS:0080, dann wieder von CS:0100 ab in den Speicher schreiben, wären alle Befehle, die feste Adressen erlauben, unmöglich.

Der Stapelzeiger SP zeigt mit FFFE (bzw. FFEE) auf das letzte Wort (bzw. auf eines der letzten Wörter) im Segment SS (das unter den gemachten Voraussetzungen noch gleich CS ist). Es wurde bereits darauf hingewiesen, daß bei Manipulationen mit SP, die in gleicher Art wie Manipulationen mit AX oder SI erfolgen können, besondere Vorsicht am Platze ist; denn SP wird von einigen Befehlen automatisch nach einem bestimmten Grundsatz

beeinflußt. Diese Befehle sind Unterprogrammaufruf, CALL..., Rückkehr von einem Unterprogramm, RET bzw. IRET, dazu die systemeigenen Unterprogramme, die mit INT... aufgerufen werden, und die eigentlichen Stapelbefehle PUSH... (PUSHF) und POP... (POPF); dazu im folgenden einige grundsätzliche Bemerkungen.

a. Unterprogramme

Ein Unterprogramm ist eine Befehlsabfolge, die in gleicher Form öfters im Programm gebraucht wird. Anstatt diese an jeder dieser Stellen von neuem einzufügen, wird sie als unabhängiges Programmstück z.B. an das Ende des ganzen Programms gestellt, mit einer Marke versehen und mit dem Befehl RET abgeschlossen. Die ganze Befehlsfolge wird im Hauptprogramm, wo sie gebraucht wird, mit dem Befehl CALL /../ "aufgerufen"; nach der Durchführung der darin enthaltenen Befehle kehrt das System durch den Befehl RET zu dem Befehl des übergeordneten (des Haupt-) Programms zurück, der dem CALL-Befehl folgt. Dieser CALL-Befehl bewirkt nämlich nicht nur einen Sprung zu der angegebenen Marke - das würde auch der Befehl JMP /../ erledigen -, sondern zugleich auch folgendes: der Stapelzeiger SP wird um 2 vermindert und an diese Stelle des Stapels wird die Adresse des Befehls abgelegt, die dem CALL-Befehl folgt; die der Marke /../ entsprechende Adresse wird in das IP-Register übertragen, was einen Sprung zum entsprechenden Befehl - hier den Beginn des Unterprogramms - bedeutet. Durch den Befehl RET am Ende des Unterprogramms wird der Inhalt des Stapelplatzes, auf den SP zeigt, in das IP-Register übertragen, und anschließend SP um 2 vermehrt. Wenn also keine (falschen) Manipulationen mit dem Stapelzeiger während des Unterprogrammablaufs gemacht wurden, wird das übergeordnete (das Haupt-)Programm nach dem CALL-Befehl fortgeführt und der Stapelzeiger zeigt wieder auf den ursprünglichen Wert. - Ähnlich funktioniert der Befehl CALL L,... bzw. CALL FAR ... als "langer" Aufruf eines Unterprogramms, wobei nicht nur ein neues IP sondern auch ein neues

CS festgelegt und dementsprechend zunächst das für diesen CALL-Befehl noch zuständige CS und dann das IP des auf diesen Befehl folgenden Befehls im Stapel abgelegt werden (SP ist insgesamt um 4 vermindert). Durch den entsprechenden Rückkehrbefehl RET L bzw. RETF am Ende dieses Unterprogramms wird zunächst das "alte" IP und das "alte" CS zurückgeholt, SP um 4 vermehrt und so das übergeordnete Programm mit dem auf den CALL-Befehl folgenden Befehl weitergeführt.

Das System selbst stellt Unterprogramme - die Interrupts oder Unterbrechungsanforderungen - zur Verfügung, die im wesentlichen den Umgang mit der "Peripherie", mit der "Umgebung" des Prozessors verwalten; dazu gehören alle Arten von Ein- und Ausgabe: Tastatur, Bildschirm, Disketten, Drucker und ähnliches, aber auch Datums- und Uhrzeitverwendung. Diese Unterprogramme werden nach bestimmten Vorbereitungen einiger Register (siehe dazu den Abschnitt über die wichtigsten Interrupts in Kap. VII) mit dem Befehl INT, gefolgt von einer höchstens zweistelligen Hexadezimalzahl, dem sog. Interrupt-Vektor, aufgerufen; auf der Spitze des Stapels werden nicht nur CS und IP des auf den INT-Befehl folgenden Befehl des aufrufenden Programms abgelegt, sondern auch - als erstes - das Statusregister, so daß dieses nach der Rückkehr vom Interrupt-Programm (mit dem Befehl IRET) im Hauptprogramm unverändert zur Verfügung steht.

Statt des Ausdrucks: "SP wird um 2 vermindert und in den dadurch festgelegten Speicherplatz des Stapels" wird etwas abgelegt, d.h. wird dorthin **kopiert**, hat sich die Redewendung: etwas wird "an der Spitze des Stapels" abgelegt, durchgesetzt. Der umgekehrte Vorgang wird mit "von der Spitze des Stapels holen" bezeichnet. Wichtig dabei ist, daß hier nicht nur ein Kopiervorgang in und vom Stapel, sondern daß zugleich auch eine Veränderung des Stapelzeigers SP erfolgt.

b. Stapelbefehle

In ähnlicher Weise werden die Befehle PUSH... und POP ausgeführt: mit PUSH wird das im Operanden angegebene Register bzw. der dort angegebene Speicherplatz "an der Spitze des Stapels abgelegt", mit POP der Wert "von der Spitze des Stapels" in das im Operanden angegebene Register bzw. den Speicherplatz "geholt".

Diese Stapelbefehle dienen vor allem zur schnellen, vorübergehenden "Zwischenspeicherung" von Registerwerten; "vorübergehend" deswegen, weil durch folgende POP-, RET-, CALL- und PUSH-Befehle die früher im Stapel gespeicherten Werte überschrieben werden.

Bei dieser "Zwischenspeicherung" ist eine Regel wichtig, deren Begründung leicht zu sehen ist:

> Was **zuletzt** im Stapel abgelegt wurde, wird als **erstes** wieder geholt.

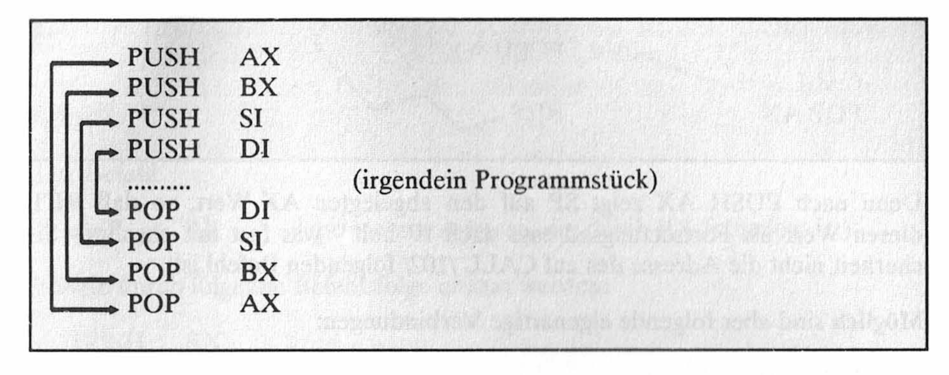
Dabei ist nicht festgelegt, daß dies in **dasselbe** Register bzw. in **denselben** Speicherplatz zurückgeholt werden muß.

Beispiele:

Wenn ein zwischenzeitlich verändertes DS wieder an CS angeglichen werden sollte, so geschieht dies am leichtesten mit der Befehlsfolge

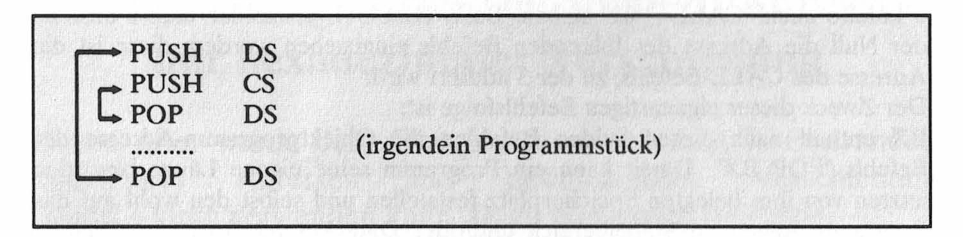
```
PUSH    CS
POP     DS
```

Sollten Register vorübergehend anderweitig gebraucht werden, ihr derzeitiger Wert aber anschließend wieder zur Verfügung stehen, dann programmiert man z.B.

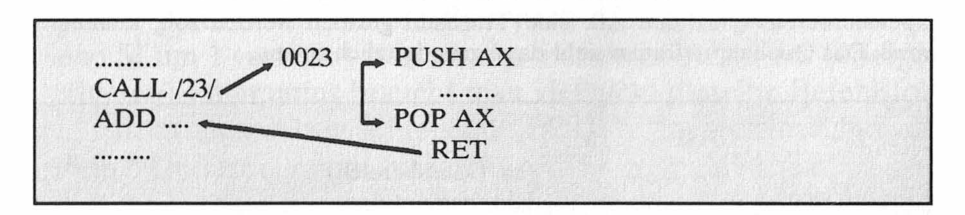


Man beachte die Reihenfolge: DI wird zuletzt im Stapel angelegt, also als erstes wieder geholt usw.

DS soll nur für ein kurzes Programmstück gleich CS sein:

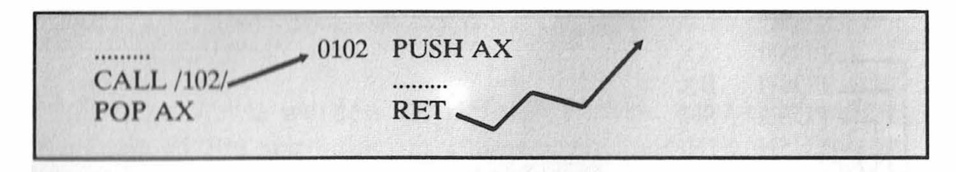


Benutzung von CALL- und PUSH-Befehlen:



Weil alles, was im Laufe des Unterprogramms im Stapel abgelegt wurde (hier nur AX), auch von dort wieder geholt wurde, führt RET zu dem (hier nicht genauer bestimmten) Befehl, der auf CALL folgt.

Falsch wäre es, wollte man ein AX, das im Verlauf des Unterprogramms im Stapel abgelegt wurde, nach der Rückkehr durch RET im übergeordneten Programm durch POP AX wieder zurückholen:



Denn nach PUSH AX zeigt SP auf den abgelegten AX-Wert, so daß RET **diesen** Wert als Fortsetzungsadresse nach IP holt - was fast mit absoluter Sicherheit nicht die Adresse des auf CALL /102/ folgenden Befehl ist!

Möglich sind aber folgende eigenartige Verbindungen:

```
CALL  0
POP   BX
```

Der Befehl "CALL 0" ist nur bei dem hier besprochenen Assembler möglich und bedeutet: der nächste Befehl ist der erste des Unterprogramms; statt CALL 0 könnte auch "CALL +01" stehen. Beim DEBUG-Assembler müßte anstelle der Null die Adresse des folgenden Befehls eingegeben werden; diese ist die Adresse des CALL-Befehls, zu der 3 addiert wird.
Der Zweck dieser eigenartigen Befehlsfolge ist:
BX enthält nach diesen beiden Befehlen die Objektprogramm-Adresse des Befehls "POP BX". Damit kann ein Programm seine eigene Länge bzw. den letzten von ihm belegten Speicherplatz feststellen und selbst den wohl auf das Programm folgenden Stapelbereich und/oder Datenbereich festlegen. Dies ist besonders in der Phase der Programmentwicklung sehr angenehm, wenn man sich nicht nach jeder Korrektur um eine neue Festlegung des Stapels oder des Speicherbereichs, an den z.B. eine Textdatei geladen werden soll, kümmern muß. Das Quellenprogramm sieht dann grundsätzlich so aus:

```
1 :       JMP    /1/
          ...............   (ev. Datenbereich)
  :0002   ...............   Hier beginnt das
          ...............   eigentliche Programm
          ...............
  :0001   CALL   0
  :       POP    BX
  :       ADD    BX,+7
  :       JMP    /2/
```

Die vier Zeilen CALL 0 bis JMP /2/ sind immer die letzten Zeilen des Quellenprogramms. Wie man sich leicht überzeugen kann, enthält BX zum Beginn des

eigentlichen Programms bei Marke 0002 die Adresse des ersten freien Speicherplatzes hinter dem Programm. Dorthin kann das Programm (durch die Berechnung von SP aus diesem BX) seinen Stapel legen und/oder eine Datendatei laden.

Der Befehl

```
JMP    BX        (springe zur Adresse, die durch BX festgelegt ist)
```

könnte durch folgende Befehlsfolge ersetzt werden:

```
PUSH   BX
RET
```

2. Anwendungen von (dezimalen) Adressen mit hexadezimaler Verschiebung

Schon öfter wurde darauf hingewiesen, daß man in Operanden (dezimale) Adressen mit hexadezimaler Verschiebung /opqr/ + klmn verwenden kann. Dazu nun einige Beispiele.

Ein Unterprogramm (UP) beginne mit LODSB, d.h. aus dem Speicherplatz DS:SI soll ein Byte nach AL kopiert und anschließend SI um 1 vermehrt werden. An manchen Stellen des (übergeordneten) Programms braucht man vielleicht dieselbe Befehlsfolge, jedoch ohne dieses einleitende LODSB. Sollte man dafür ein zweites Unterprogramm zurechtlegen? Es geht einfacher:

```
0756   LODSB
       .....             Unterprogramm (UP)
       RET
       .....
       CALL/756/         Aufruf des UP mit LODSB
       ....
       CALL/756/+1       Aufruf des UP ohne LODSB
       ....
```

Im zweiten Fall berechnet der Assembler die Adresse der Marke 0756 und erhöht sie um 1; die im Befehl angegebene Adresse verweist also auf den Befehl hinter LODSB, weil dieses in der Maschinensprache und damit im Objektprogramm **genau ein Byte** belegt; der Befehl LODSB wird also bei **diesem** Aufruf des UP übergangen.

Wie man jetzt schon sieht, ist diese Möglichkeit der Adressierung nur dann voll auszuschöpfen, wenn man die Länge der Befehle in Maschinensprache kennt. Dafür ist die schematische Angabe des Maschinencodes der Befehle in Abschnitt VI gedacht. Will man sich die Mühe des Nachzählens ersparen, kann man die zu untersuchenden Befehle bzw. Befehlsfolgen unter DEBUG mit dessen Assembler (Befehl a) einschreiben und dann die (relative) Lage der angepeilten Byte unmittelbar dem Bildschirm entnehmen.

Ein weiters Beispiel:

In einem verzweigten Programm sei es prinzipiell möglich, daß ein UP auf verwickelten Wegen sich selbst aufruft und damit eine nicht abbrechende Wiederholung erzeugt würde. Das UP sollte sich also, einmal aufgerufen, selbst sperren, und erst unmittelbar, bevor es durch RET in das übergeordnete Programm zurückverweist, sich selbst wieder freigeben. Dies gelingt so:

```
0011    NOP
        CS:
        MOV BY [/11/],C3        C3 ist der Code für RET
        .......
        .......
        CS:
        MOV BY [/11/],90        90 ist der Code für NOP
        RET
```

Wenn dieses UP aufgerufen wird, beginnt es mit NOP, No-Operation. Die nächsten beiden Befehlszeilen gehören zusammen:

"CS:" ist ein "Präfix", ein Vorsatz-Befehl, der nur besagt, daß die Adresse im nächsten Befehl mit CS berechnet werden soll; da das UP im Code-Segment liegt, liegt natürlich auch der mit Marke 0011 bezeichnete Speicherplatz, der zunächst mit 90 belegt ist, in CS. (Der der Marke 0011 entsprechende Hexadezimalwert, den der Assembler ausrechnet, zeigt, ohne dieses Präfix mit DS verbunden - s. oben S. 25 -, ganz woanders hin, wenn DS nicht gleich CS ist.) Der MOV-Befehl legt also an die Stelle des Byte 90 ein Byte C3. Wenn also je - solange dieses UP abgewickelt wird - in einem weiteren Unterprogramm der Befehl CALL /11/ vorkäme, würde dieses RET in 0011 den sofortigen Rücksprung bedeuten - unser UP kann nicht ein zweites Mal aufgerufen werden. Erst bevor es seinerseits wieder mit RET in das aufrufende Programm zurückgibt, hebt es diese Sperrung wieder auf.

Ein weiteres Beispiel: Die Befehlsfolge - eine sogenannte Interrupt-Funktion

```
MOV   AH,9
MOV   DX,/../
INT   21
```

bewirkt, daß die (Text-)Zeichenfolge von DS:DX an, bis dort das Zeichen $ (das Byte 24) auftaucht, auf dem Bildschirm ausgegeben wird. So müssen also drei Register, AH, DX und DS, vorübergehend für die Ausgabe festgelegt werden. Bei jeder Stelle des Programms, an der eine Ausgabe erfolgen sollte, muß also auch überlegt werden, ob DS jetzt gerade richtig ist und ob man den jetzigen Wert von AH oder von DX nach der Ausgabe vielleicht doch noch braucht. Um sich diese Überlegungen bei jeder Bildschirmausgabe zu sparen, macht man folgendes Unterprogramm:

```
0004  PUSH  DX
      MOV   DX,0
      PUSH  AX
      PUSH  DS
      PUSH  CS
      POP   DS
      MOV   AH,9
      INT   21
      POP   DS
      POP   AX
      POP   DX
      RET
```

Die Texte, die auf dem Bildschirm erscheinen sollen, sind natürlich, mit bestimmten Marken versehen - z.B. 0012, 0020 -, in unserem Programm enthalten und werden mit dem Programm zusammen von DOS im Programmsegment geladen. Deswegen muß das DS, das beim Aufruf des UP gegeben ist, vorübergehend im Stapel abgelegt (PUSH DS leistet dies) und dann gleich CS gemacht werden (PUSH CS - POP DS). AX und DX werden ebenso im Stapel "aufbewahrt". Im UP muß aber noch DX festgelegt werden: PUSH DX ist im Maschinencode 1 Byte lang, MOV DX,... beginnt mit dem Byte BA, dem dann die gewünschte Adresse (2 Byte lang) folgt; so muß diese Adresse vor Aufruf des UP in das zweite Byte hinter dem Beginn des UP abgelegt werden:

```
.......
(CS:)
MOV    WO [/4/+2],/12/
Call   /4/
......
(CS:)
MOV    WO [/4/+2],/20/
Call   /4/
```

Vor dem ersten Aufruf des UP wird die Adresse /12/ so in den Befehl MOV DX,.. abgelegt, daß durch den Aufruf des UP der

Text auf dem Bildschirm erscheint, der bei Marke 0012 beginnt; vor dem zweiten Aufruf entsprechend für /20/ bzw. 0020. Das Präfix "CS:" sollte man eingeben, wenn an dieser Stelle des Programms auch nur möglicherweise DS nicht gleich CS ist.

Ein letztes Beispiel in diesem Zusammenhang: Die bedingten Befehle lassen nur zwei Möglichkeiten zu, entweder Sprung oder Fortfahren mit dem nächsten Befehl. Wenn das Programm also an einem Punkt, z.B. abhängig von der Zahl in AL, an verschiedenen Stellen fortgeführt werden soll, könnte man so vorgehen:

```
CMP   AL,0
JNZ   +02
JMP   /345/
CMP   AL,1
JNZ   +02
JMP   /24/
CMP   AL,2
JNZ   +02
JMP   /121/
usw.
```

Übersichtlicher wird das Programm so:

Zunächst eine Vorbemerkung:

JMP BX heißt: Sprung zur Adresse **in CS**, die durch BX angegeben ist;

JMP [BX] heißt: Sprung zu der Adresse **in CS**, die in dem **Speicherplatz** abgelegt ist, der durch **DS**:BX bezeichnet wird.

Nun das Programm:

```
        MOV    BL,AL
        SUB    BH,BH
        SHL    BX
        (CS:)
        JMP    [BX+/7/]
0007    DB     /345//24/ /121/ ...
        DB     /.../ ...
```

AL wird nach BL kopiert, BH gleich Null gemacht, somit steht nun im Register BX die entscheidende Zahl. Alle Adressen sind im Maschinencode zwei Byte lang, so daß die gewünschten Adressen von dem durch 0007 markierten Byte an je im Zweierabstand hintereinanderstehen; bei AL gleich 1 und - zunächst - BL ebenso gleich 1 würde der Sprung JMP [BX+/7/] an die Adresse erfolgen, die in Speicherplatz 1+/7/ (bzw. in der gewohnten Schreibweise /7/+1) beginnt: dies wäre faktisch eine Adresse, die aus dem zweiten Byte der ersten Adresse (die 0345 entspricht) und dem ersten Byte der richtigen Adresse (die 0024 entspricht) zusammengesetzt, also sicher falsch ist. Deshalb muß BX vor dem Sprung verdoppelt werden, und dies geschieht durch den Befehl SHL BX. Die Adressen in der DB-Zeile können aneinandergefügt oder durch Leerstellen getrennt, oder schließlich in einer neuen DB-Zeile (ohne neue Markierung!) fortgesetzt werden. Es muß nur garantiert sein, daß für jede Zahl in AL, von Null bis zur (an dieser Stelle des Programms) größtmöglichen, eine Adresse in der (den) DB-Zeile(n) vorgesehen ist. Das Präfix "CS:" ist nur notwendig, wenn an dieser Stelle DS nicht gleich CS ist; wenn man dies beim Programmieren nicht umständlich nachprüfen will, fügt man "CS:" ein, es schadet nichts.
Beim Beispiel verzweigt das Programm bei AL=0 durch den JMP-Befehl zu Marke 0327, bei AL=1 zu Marke 0024, bei AL=2 zu Marke 0121 und so fort.

V. Programmbeispiele

Das simple Programm, das auf den ersten Seiten angeführt wurde, hat die in AL eingegebene Zahl dadurch "erraten", daß gefragt wurde, ob diese Zahl gleich 0 sei, wenn nicht, ob gleich 1, wenn auch dies nicht, ob gleich 2 usw., bis die Zahl gefunden war. Wurde in AL z.B. 02 eingegeben, wäre die Zahl bei der dritten Abfrage gefunden, wurde aber FF, das ist dezimal 255, eingegeben, dann erst bei der 256ten Abfrage. In diesen Größenordnungen spielt die Zeit, die das Programm für die Suche braucht, noch keine Rolle. Dies mag sich aber bei der Verwendung eines größeren Zahlenbereiches und vielleicht mit weiteren Aufgaben des Programms, die mit jeder einzelnen Zahl verbunden sind, wesentlich ändern. Noch dazu werden kleine Zahlen sehr viel schneller gefunden als große, obwohl sie bei der Eingabe "irgendeiner" Zahl von 00 bis FF gleich wahrscheinlich sind. Das Programm sollte "systematischer" sein:

So lassen wir das Programm zunächst abfragen, ob die Zahl in AL in der oberen Hälfte des möglichen Zahlenbereichs liegt; wenn z.B. "ja", so ist nur noch dieser obere Zahlenbereich möglich - und wir lassen wieder fragen, ob die Zahl in AL in der oberen Hälfte des (noch) möglichen Zahlenbereichs liegt usw.; wenn "nein", dann ist nur noch der untere Zahlenbereich möglich - die weitere Abfrage ist **dieselbe**. Offensichtlich erreichen wir durch diese Einengung des möglichen Bereiches ("Intervallschachtelung") jede Zahl des möglichen Bereiches gleich schnell.

Eine erste Ausführung dieses Programms:

```
MOV    BL,80
CMP    AL,BL
JA     +03       Sprung, wenn AL größer als BL ist.
SUB    BL,40
JMPS   +02
ADD    BL,40
CMP    AL,BL
JA     +03
SUB    AL,20
JMPS   +02
ADD    BL,20
CMP    AL,BL     usw.
       .........
CMP    AL,BL
JA     +03
SUB    AL,1
JMPS   +02
ADD    AL,1
CMP    AL,BL
JA     +02
JMPS   +02
ADD    bl,1
......           irgendeine Fortsetzung des Programms.
                 Nun ist in jedem Fall BL gleich AL, die
                 Zahl ist "gefunden".
```

Bei diesem Programm fällt auf, daß siebenmal dieselbe Befehlsfolge vorkommt, jedesmal aber ist die Zahl, die addiert oder subtrahiert werden soll, anders, so daß keine Wiederholung der Folge, keine "Schleife", möglich scheint; jedoch ist der folgende Wert dieser Zahlen immer die Hälfte des vorausgegangenen: 40, 20, 10, 08, 04, 02, 01. Diese Zahlen sehen im binären Code so aus:

0100 0000, 0010 0000, 0001 0000, 0000 1000, usw.,

das besetzte Bit ist jeweils um eine Stelle nach rechts verschoben. Um eine solche Rechtsverschiebung zu bewerkstelligen, gibt es

den Befehl SHR... So liegt es nahe, die zu addierende bzw. subtrahierende Zahl in einem anderen Register jeweils durch einen solchen Schiebebefehl zu erzeugen und eine Schleife zu bilden:

```
MOV     CX,7
MOV     BL,80
MOV     BH,40
CMP     AL,BL
JA      +03
SUB     BL,BH
JMPS    +02
ADD     BL,BH
SHR     BH
LOOP    -06
CMP     AL,BL
JA      +02
JMPS    +02
ADD     BL,1
.....
```

Die letzen drei Befehle könnten durch folgende zwei ersetzt werden:

```
JZ      +02        Sprung bei Gleichheit
ADD     BL,1
.....,
```

Das Programm ist nun etwas länger als das ursprüngliche, ist aber für die zu suchende Zahl FF genauso schnell, wie für 08 - und für alle Zahlen zwischen (etwa) 08 und FF schneller als das ursprüngliche: bei jeder Suche werden nur acht Abfragen benötigt, um die Zahl zu finden, während beim ursprünglichen Programm für alle Zahlen, die größer als 8 sind, mehr, u.U. sogar wesentlich mehr Abfragen und damit kostbare Zeit benötigt werden.

Ein weiteres Beispiel: In einem Text, der bei DS:0 beginnt, soll

ein bestimmtes Wort, z.B. "Tag" gesucht werden.

```
PUSH   DS
POP    ES       ES ist nun gleich DS
MOV    DI,0     ES:DI zeigt auf den Anfang des Textes
MOV    AL,54    in AL ist nun der Code für T; auch: mov al,"T"
SCASB           Vergleicht Byte in ES:DI mit AL, und DI+1
JNE    -01      bei "ungleich" zurück und Vergl. mit folg., sonst:
MOV    AL,"a"   auch: mov al,61
SCASB
JNE    -05      wenn nicht "Ta", nächstes "T" suchen, sonst:
MOV    AL,"g"   auch: mov al,67
SCASB
JNE    -08      wenn nicht "Tag", nächstes "T" suchen, sonst:
..........      irgendeine Fortsetzung
```

Dieses Programmstück "funktioniert", ist aber noch sehr unbefriedigend: Für jedes zu suchende Wort müßte in dieser Art ein Programm geschrieben werden; DI zeigt nach erfolgreichem Suchen nicht an den Anfang des Wortes "Tag", sondern dahinter; schließlich, was passiert, wenn der Text zufällig das Wort "Tag" nicht enthält?

Lösen wir zunächst die Aufgabe, feststellen zu lassen, ob ein Wort des Textes ein zweites Mal vorkommt.

```
PUSH   DS
POP    ES
MOV    DI,0
MOV    SI,opqr   opqr zeige auf das zu suchende Wort
LODSB            der erste Buchstabe dieses Wortes ist in AL, SI+1.
SCASB            ist Byte in ES:DI gleich AL? - DI+1
JNE    -01
LODSB            der zweite Buchstabe des Wortes in AL, SI+1
SCASB
JNE    -06       SI muß bei ungleich wieder auf Wortanfang zeigen
LODSB
SCASB
JNE    -09
.........        usw. "je nach Länge des Wortes".
```

Dieses Programmstück ist sicher "erfolgreich"; denn sobald DI gleich SI geworden ist, zeigen beide auf **dieselbe** Zeichenfolge. Wenn also dieses gleiche Wort nicht schon vorher im Text einmal vorkam, muß das Programm von der vierten Zeile an neu gestartet werden, damit eine **zweite** gleiche Zeichenfolge gefunden werden kann. - Die Befehle LODSB und SCASB können in den einen Befehl CMPSB ersetzt werden. Nur das erste Paar LODSB-SCASB muß getrennt stehen bleiben, weil hier bei Ungleicheit allein DI um 1 vergrößert werden soll; der durch das hier stehende LODSB nach AL kopierte erste Buchstabe des zu suchenden Wortes soll ja bleiben! Für die weiteren Zeichen des Wortes ist offensichtlich eine Schleife möglich:

```
1:  PUSH   DS
2:  POP    ES
3:  MOV    DI,0
4:  MOV    SI,opqr
5:  MOV    CX,....    CX soll die hex-Länge des Wortes angeben.
6:  LODSB
7:  SCASB
8:  JNE    -01
9:  DEC    CX         CX um 1 kleiner, weil das erste Zeichen gefunden!
10: CMPSB
11: LOOPE  -01        solange gleich zurück, höchstens CX-mal
12: JNZ    -08        wenn ungleich, mit gleichem SI und CX von vorne!
13: .......
```

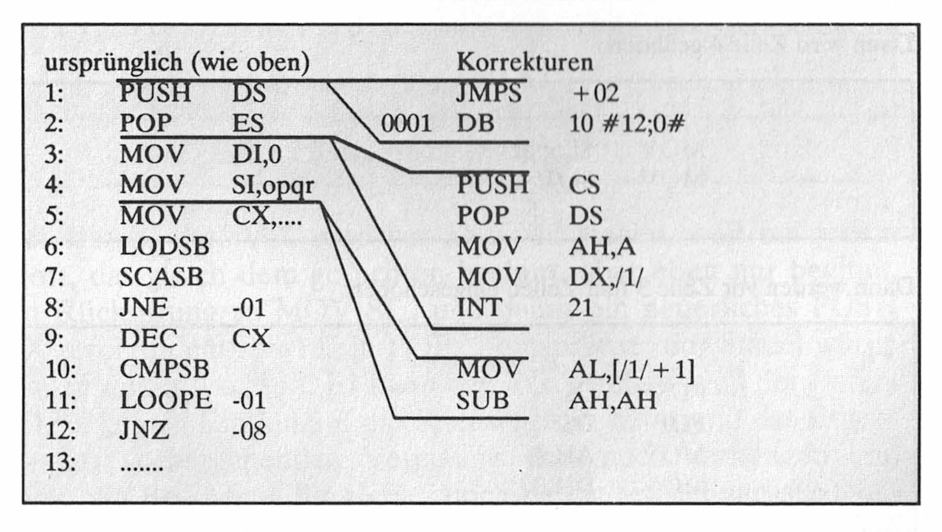
Ein weiterer Schritt: Ein beliebiges Wort soll gesucht werden, das von der Tastatur eingegeben wird. Dazu brauchen wir eine Eingabe-Funktion, ein "INT", das die "Eingabe einer Zeichenkette" erlaubt. Für diesen Interrupt müssen wir einen "Puffer", eine Serie von Speicherplätzen, in folgender Art vorbereiten: im ersten Byte wird festgelegt, wie lange die einzugebende Zeichenkette sein darf - diese Zahl sei x; dann folgen x+2 reservierte Byte. Zum Aufruf dieser Eingabe-Funktion muß AH gleich A (nicht "A", sondern die hexadezimale "zehn") gesetzt sein und

DS:DX auf den Beginn dieses Puffers zeigen; durch den dann folgenden Befehl INT 21 wird das System veranlaßt, auf die Eingabe einer Zeichenkette zu warten; diese Eingabe wird mit einer Zeilenschaltung (= Enter) abgeschlossen. Anschließend (zum Zeitpunkt des auf INT 21 im Programm folgenden Befehls) sieht der Puffer so aus: im ersten Byte ist die erlaubte Zeichenlänge unverändert geblieben; im zweiten Byte ist die **Länge der tatsächlich eingegebenen Zeichenkette** eingetragen. Ab dem dritten Byte sind die Codes der eingegebenen Zeichen abgelegt; diese Kette endet mit dem (im zweiten Byte nicht mitgezählten) Byte 0D, das der Zeilenschaltung entspricht. (Wir setzen wieder voraus, daß zunächst DS:0 auf den zu untersuchenden Text weise. Die eingeklammerten Zeilennummern verweisen auf die Zeilen der vorhergehenden Programmform; der Zusatz "ä" bedeutet, daß die Zeile geändert werden muß.)

```
       1:      JMPS   +02           Überspringen des Puffers
       2:0001  DB     10 #12;00#    Eingabe von sechzehn Zeichen möglich
    (1)3:      PUSH   DS
    (2)4:      POP    ES            ES:0 zeigt auf Beginn des Textes
       5:      PUSH   CS
       6:      POP    DS            DS ist gleich CS
       7:      MOV    AH,A
       8:      MOV    DX,/1/        DS:DX zeigt auf Puffer
       9:      INT    21            Eingabe einer Zeichenfolge (=ZF)
   (3)10:      MOV    DI,0          ES:DI zeigt auf Beginn des Textes
  (4ä)11:      MOV    SI,/1/+2      DS:SI zeigt auf Beginn der eingegeb. ZF
      12:      MOV    AL,[/1/+1]    in AL die tatsächliche Länge der ZF
      13:      SUB    AH,AH         in AX diese Länge (AH gleich 0)
  (5ä)14:      MOV    CX,AX         auch in CX diese Länge
   (6)15:      LODSB
   (7)16:      SCASB
   (8)17:      JNE    -01
   (9)18:      DEC    CX
  (10)19:      CMPSB
  (11)20:      LOOPE  -01
 (12ä)21:      JNE    -10
      22:      .......
```

Eine Zwischenüberlegung:
Wie werden Quellenprogramme mit EDLIN korrigiert?

Hier soll ein Kunstgriff angeführt werden, der die Korrektur des Quellenprogramms in EDLIN vereinfacht: Man beginnt mit der Korrektur **von hinten** - aus folgendem Grund: Da bei Korrekturen im allgemeinen Zeilen gelöscht bzw. neue Zeilen eingefügt werden müssen, ändert sich die Numerierung der bereits vorhandenen nachfolgenden Zeilen. Im Beispiel muß die ursprüngliche Zeile 12 (JNE -08) geändert werden (JNE -10); macht man dies, **nachdem** die neuen Zeilen 5 - 9 eingefügt wurden, muß man **suchen**, welche Zeilennummer nun die Zeile "JNE -08" hat; im Beispiel ist dies noch einfach, bei größeren Programmen kann ein gleichlautender Befehl öfters vorkommen und es kann sehr umständlich werden, festzustellen, welcher von diesen geändert werden muß. Wenn man aber bei der höchsten Zeilennummer mit der Korrektur beginnt, behalten alle noch zu korrigierenden Zeilen (die ja **davor** stehen), ihre bisherige Zeilennummer. Dies soll einmal ausführlich gezeigt werden:



Zur Korrektur einer Zeile wird nach der EDLIN-Systemanfrage einfach die Zeilennummer eingegeben; das System zeigt diese Zeile an und wartet auf eine Neueingabe dieser Zeile (eventuell unter Verwendung der DOS-Korrekturtasten F1 bis F6):

```
*12
        12:     JNZ     -08
        12:     JNZ     -10
*
```

Dann wird die vorhergehende Korrektur gemacht, hier Zeile 5:

```
*5
        5:          MOV     CX,....
        5:          MOV     CX,AX
*
```

Dann werden vor Zeile 5 zwei Zeilen mit dem Insert-Befehl eingeschoben:

```
*5I
        5:          MOV     AL,[/1/+1]
        6:          SUB     AH,AH
        7:  ^C      (mit CTRL-Break Abbruch der Eingabe)
*
```

Dann wird Zeile 4 geändert:

```
*4
        4:          MOV     SI,opqr
        4:          MOV     SI,/1/+2
*
```

Dann werden vor Zeile 3 fünf Zeilen eingeschoben:

```
*3I
        3:          PUSH    CS
        4:          POP     DS
        5:          MOV     AH,A
        6:          MOV     DX,/1/
        7:          INT     21
        8:  ^C
*
```

Dann noch vor Zeile 1 zwei Zeilen:

```
*1I
        1:          JMPS    '+02
        2:0001      DB      10 #12;0#
        3:  ^C
*
```

Wenn man sich jetzt mit dem Befehl 1,#L das ganze Programm auflisten läßt, werden die 21 Zeilen des korrigierten Programms wie oben gezeigt. Nun wieder zurück zur Weiterführung unseres Programms!

Nun soll noch DI nach erfolgreicher Suche auf den Beginn des Wortes im Text zeigen:

Nach dem Befehl MOV CX,AX fügen wir ein:

```
15:   PUSH    AX       Vorübergehende Ablage von AX im Stapel
16:   LODSB
17:   SCASB
      .....
21:   LOOPE   -01
22:   POP     AX
23:   JNE     -12
24:   SUB     DI,AX    die Wortlänge wird von DI subtrahiert,
                       DI zeigt zum Wortanfang.
25:   .......
```

Der Befehl POP AX **muß** vor JNE -12 stehen, weil bei jedem Wort, das gleich dem gesuchten **beginnt**, aber eben nur beginnt, ein Rücksprung zu MOV SI,.. und damit ein neuerliches PUSH AX erfolgt, dem sonst kein POP ... entspräche: der Stapel würde immer weiter wachsen! Er **kann** vor JNE.. stehen, weil der Befehl POP .. keinen Einfluß auf das Statusregister hat, somit das Ergebnis des vorhergehenden Vergleichs dort noch vorhanden und damit die Bedingung für JNE ... noch richtig feststellbar ist. - Nun muß noch die Textlänge berücksichtigt werden: Bei den Dateien, die im allgemein für PCs üblichen Code, dem ASCII-Code (American Standard Code for Information Interchange) geschrieben sind, ist das Textendezeichen 1A. Nehmen wir der Einfachheit halber an, daß wir keinen Text zu untersuchen haben, der mitten in einem Wort abbricht, so kann dieses Zeichen nicht auftreten, wenn unser Programm bei CMPSB den Wortvergleich (nach dem ersten gleichen Buchstaben) ausführt. Also kann nur bei SCASB weiter oben ein Vergleich von AL mit dem Textendezeichen an-

gebracht sein. Wir fragen also zunächst, ob ES:DI etwa auf dieses Zeichen zeige, dann erst lassen wir mit SCASB den Vergleich mit AL ausführen und DI um 1 erhöhen:

```
        15:       .....
(16)    16:       LODSB
        17:       ES:
        18:       CMP     BY [DI],1A
        19:       JZ      /2/
(17)    20:       SCASB
(18ä)   21:       JNE     -04
        22:       DEC     CX
        23:       CMPSB
        24:       LOOPE   -01
        25:       POP     AX
(23ä)   26        JNE     -15        die Zeilenzahl muß erhöht werden!
        27:       SUB     DI,AX
        28:       ......             Programmfortsetzung, wenn gefunden
        29:0002   ......             Programmfortsetzung,
                                     wenn nicht (mehr) gefunden
```

Dieses Programm kann nun natürlich weiter ausgebaut werden: z.B. könnte die Fortsetzung nach erfolgreichem Suchen etwa DI auf dem Bildschirm anzeigen oder vielleicht sogar (Seiten-) und Zeilenzahl angeben, wo das Wort vorkommt, weiter anfragen, ob im Text weitergesucht oder die Suche abgebrochen werden soll (bei Weitersuchen sollte man nicht vergessen, DI um 1 zu erhöhen, weil sonst die "weitere" Suche an **derselben** Stelle erfolgt und damit das Programm "hängen" bleibt). Bei erfolglosem Suchen - ab der Marke 0002 - könnte dies am Bildschirm angezeigt, dann die Frage gestellt werden, ob die Suche eines anderen Wortes gewünscht sei; wenn ja, muß ein Sprung zum Beginn des Programms erfolgen, sonst wird es mit INT 20 (= endgültiger Programmabbruch) beendet. Diese und ähnliche Erweiterungen seien hier jedoch nicht genauer ausgeführt.

Nur eine Ergänzung soll noch besprochen werden: Bisher hatten wir vorausgesetzt, daß der zu untersuchende Text in einem

Segment Platz habe, also nicht länger als 65.536 Zeichen sei. Diese doch sehr einschränkende Voraussetzung kann umgangen werden, wenn man bei wachsendem DI sozusagen ES immer wieder "nachholt". Unbesorgt können wir DI bis etwa FF00 wachsen lassen - es gibt wohl kein Wort, das mehr als 250 Zeichen hat und von FF00 an über das Segmentende (FFFF) hinausgehen würde: wir müssen also die Abfrage über die Größe von DI nicht auch während eines Wortvergleichs (im Bereich von SCASB) machen, es genügt, wenn wir dies jedesmal an einem Wortanfang, also etwa nach MOV CX,AX machen. Zugleich erinnern wir uns, wie ein Speicherplatz mit ES:DI (bzw. DS:SI) festgelegt wird (siehe oben S. 23/24): Bei ES gleich 14F3 und DI gleich 2587 gilt:

ES	14F3**0**
DI	2587
ES:DI	174B7

Dieser selbe Speicherplatz kann offensichtlich durch ganz verschiedene Wertepaare ES und DI festgelegt werden:

ES	13E2**0**	ES	1000**0**	ES	174B**0**
DI	3697	DI	74B7	DI	0007
ES:DI	174B7	ES:DI	174B7	ES:DI	174B7

Also durch ES = 13E2 mit DI = 3697, ES = 1000 mit DI = 74B7 und ES = 174B mit DI = 0007. Diese beiden letzten Fälle weisen uns auf den Weg hin, den wir einschlagen müssen, um ES bei DI gleich oder größer FF00 "nachzuholen": die vorderen drei Halb-Byte von DI (im Beispiel **74B**7) müssen als einfache Zahl (74B) zu ES addiert werden (1000 + 74B = 1**74B**), das letzte Halbbyte (7 in 74B7) verbleibt in DI (mit vorlaufenden Nullen).

```
     11:  CMP   DI,ff00
     12:  JC    +07       solange kleiner, ohne Änderung weiter
     13:  MOV   BX,DI
     14:  MOV   CL,4
     15:  SHR   BX,CL     BX um 4 Bit (=1 Halbbyte) nach rechts
     16:  MOV   AX,ES
     17:  ADD   AX,BX
     18:  MOV   ES,AX
     19:  AND   DI,F      in DI bleiben nur die letzten 4 Bit
                          (das letzte Halbbyte)
(11) 20:  ......
```

Dieses Programmstück wird nach dem Befehl MOV DI,0 eingeschoben und der bisherige Befehl JNE -15 gegen Ende des Programms muß auf CMP DI,FF00 zurückweisen, also auf JNE -24 verändert werden. Da die in diesem neuen Programmstück verwendeten Register BX, AX, CL (damit CX) anschließend, soweit gebraucht, neu definiert werden, können sie hier ohne vorübergehende "Rettung" (z.B. im Stapel) verwendet werden.

Wenn man versucht, diese Programmentwicklung jeweils mit EDLIN nachzuvollziehen, indem man die neuen Befehle einschiebt und unter Umständen bereits vorhandene löscht oder ändert, wird man leicht feststellen, wie angenehm bereits Bewährtes erhalten bleibt und wie wenig dies wegen neuer Einschübe, die hoffentlich jeweils Verbesserungen des Programms bedeuten, geändert werden muß.

Die Beispiele könnten natürlich beliebig vermehrt werden. Doch sollen die bisherigen als Anregung zu eigenen Versuchen genügen.

VI. Befehle des 8086/8088 in Assembler-Sprache und Maschinencode

Es wäre zu umständlich und unübersichtlich, alle Befehle mit ihren möglichen Operanden und dem jeweils entsprechenden Maschinencode hier aufzulisten; deshalb werden hier der Maschinencode nur schematisch dargestellt und im übrigen einige Abkürzungen verwendet:

reg	steht als Abkürzung für irgend ein Register: AX, AH, AL; BX, BH, BL; CX, CH, CL; DX, DH, DL; BP; SP; SI; DI.
segreg	steht als Abkürzung für ein Segmentregister CS, DS, ES oder SS.
(mem)	steht als Abkürzung für irgendeine der Speicherplatzbezeichnungen, die auf S. 25/26 aufgelistet sind.
FZ	steht als Abkürzung für eine Festzahl; in welchen Formen diese eingegeben werden können, ist auf S. 27 aufgezeigt.
--	kein Operand (nur das Befehlswort ist einzugeben).

In der ersten Spalte wird jeweils das Befehlswort, in der zweiten Spalte der Operand angegeben. In der dritten Spalte ist der Maschinencode schematisch angegeben; hierzu noch einige Erklärungen.

Der Maschinencode für einen Befehl besteht zunächst aus ein bis bis zwei Byte, die den Befehl mit den betroffenen Registern bzw.

Speicherplätzen festlegen; **wenn** der Speicherplatz im Operanden numerisch angegeben ist - in der Form [klmn], [/opqr/] oder [/opqr/ + klmn] - folgt dessen Adresse, zuerst das niederwertige, dann das höherwertige Byte; dann folgt, **wenn** im Operanden ein Displacement angegeben ist, das zweistellige Displacement oder zuerst das niederwertige, dann das höherwertige Byte des vierstelligen Displacements; dann folgt, **wenn** im Operanden eine Festzahl angegeben ist, diese zweistellig oder zuerst ihr niederwertiges, dann ihr höherwertiges Byte. Da sich die numerische direkte Angabe des betroffenen Speicherplatzes und die Angabe eines Displacements gegenseitig ausschließen, kann ein Befehl insgesamt höchstens sechs Byte umfassen.

Allgemein hat also ein Befehl im Maschinencode folgende Struktur:

(1 oder 2 Byte) (mnkl) (jj(ii)) (gg(ff))

klmn	ist die Adresse des betroffenen, numerisch angegebenen Speicherplatzes,
jj	ist ein zweistelliges Displacement,
iijj	ein vierstelliges Displacement,
gg	eine zweistellige Festzahl,
ffgg	eine vierstellige Festzahl.

(Man beachte die scheinbare Vertauschung des jeweils höherwertigen und niederwertigen Byte!)

Dazu einige Beispiele:

ADD AL,BL	Nur Register betroffen:
00 D8	2 Byte Maschinencode (=MC).
ADD [klmn],CX	ein Speicherplatz numerisch bezeichnet:
01 09 mn kl	2 Byte MC, vierstell. Nummer des Speicherplatzes.
z.B. ADD [1234],CX	= 01 09 34 12
ADD AL,gg	AL betroffen und Angabe einer Festzahl:
04 gg	1 Byte MC, zweistellige Festzahl.
z.B. ADD AL,12	= 04 12

ADD AX,ffgg	AX betroffen und Angabe einer Festzahl:
05 gg ff	1 Byte MC, vierstellige Festzahl.
z.B. ADD AX,1234	= 05 34 12
oder ADD AX,7	= 05 07 00
ADD DX,[SI+jj]	ein (zweistelliges) Displacement:
03 52 jj	2 Byte MC, zweistelliges Displacement.
z.B. ADD DX,[SI+12]	= 03 52 12
ADD BP,[BX+SI+iijj]	(vierstelliges) Displacement:
03 A8 jj ii	2 Byte MC, vierstelliges Displacement.
z.B. ADD BP,[BP+SI+1234]	= 03 A8 34 12
ADD BY [SI],gg	(zweistellige) Festzahl:
80 04 gg	2 Byte MC, zweistellige Festzahl.
z.B. ADD BY [SI],12	= 80 04 12
ADD BY [klmn],gg	Speicherplatz numerisch, (zweistellige) Festzahl:
80 06 mn kl gg	2 Byte MC, Speicherplatz, Festzahl.
z.B. ADD BY [1234],56	= 80 06 34 12 56
ADD BY [SI+jj],gg	(zweistell.) Displacement, (zweistellige) Festzahl:
80 44 jj gg	2 Byte MC, Displacement, Festzahl.
z.B. ADD BY [SI+12],56	= 80 44 12 56
ADD BY [SI+iijj],gg	(vierstell.) Displacement, (zweistell.) Festzahl:
80 84 jj ii gg	2 Byte MC, Displacement, Festzahl.
z.B. ADD BY [SI+123],4	= 80 84 23 01 04
ADD WO [SI+jj],ffgg	(zweistell.) Displacement, (vierstell.) Festzahl:
81 44 jj gg ff	2 Byte MC, Displacement, Festzahl.
z.B. ADD WO [SI+1],234	= 81 44 01 34 02
ADD WO [SI+iijj],ffgg	(vierstell.) Displacement, (vierstell.) Festzahl:
81 84 jj ii gg ff	2 Byte MC, Displacement, Festzahl.
z.B. ADD [SI+123],456	= 81 84 23 01 56 04

AAA -- (1 Byte)

(Ascii Adjust for Addition)
Korrektur des in AX stehenden Ergebnisses einer einstelligen ASCII- bzw. einer einstelligen ungepackt BCD-Addition. Im ASCII-Code werden die Dezimalzahlen 0 bis 9 durch die Byte 30 bis 39 dargestellt; im ungepackten BCD-Code durch die Byte 00 bis 09 (im gepackten BCD-Code nehmen die Dezimalziffern nur ein halbes Byte ein, so daß zweistellige Dezimalziffern in einem Byte dargestellt werden können).
Angenommen, in AX steht eine zweistellige ungepackte BCD-Dezimalzahl; wird nun zu AL eine ebensolche einstellige Dezimalzahl addiert (der Additionsbefehl behandelt die Byte als Hexadezimalzahlen!), so wird durch den Befehl AAA das Ergebnis in AX so korrigiert, daß es als Ergebnis der entsprechenden Dezimaladdition erscheint.
Definierter Einfluß nur auf Übertragstatusbit C und Hilfsübertragstatusbit A. C=1 nur wenn ein (dezimaler) Übertrag nach AH erfolgt ist; A=1 nur wenn bei der hexadezimalen Addition in AL ein Übertrag vom niederwertigen Halbbyte erfolgt ist.

AAD A (1 Byte)(gg)

(Ascii Adjust for Division)
Vorbereitung eines in AX stehenden ASCII- bzw. ungepackt BCD-Quotienten für eine Division; gleichbedeutend mit einer Umwandlung einer zweistelligen ASCII- bzw. ungepackt BCD-Dezimalzahl, die in AX steht, in die entsprechende ein Byte lange Hexadezimalzahl in AL.
(Durch die Änderung des Operanden können analog zweistellige Zahlen anderer Zahlensysteme in Hexadezimalzahlen umgewandelt werden, solange die Basis dieser Systeme kleiner als dez 16 = hex 10 ist).
Definierter Einfluß nur auf Statusbit für Null (Z), Vorzeichen (S) und Parität (P).

AAM A (1 Byte)(gg)

(Ascii Adjust for Multiply)
Korrektur des Registers AL, in dem das Produkt zweier einstelliger ungepackter BCD-Dezimalzahlen steht, so daß anschließend in AX die ungepackte BCD-Dezimalzahl als Ergebnis steht; gleichbedeutend einer Umwandlung einer Hexadezimalzahl kleiner als 64, die in AL steht, in eine ungepackte BCD-Dezimalzahl in AX.

(Durch die Änderung des Operanden können analog Hexadezimalzahlen in AL in Zahlen anderer Zahlensysteme in AX umgewandelt werden, solange die Basis dieser Systeme kleiner als dez 16 = hex 10 ist).
Definierter Einfluß nur auf Statusbit für Null (Z), Vorzeichen (S) und Parität (P).

AAS	--	(1 Byte)	

(Ascii Adjust for Subtraction)
Die zu AAA analoge Operation für **Subtraktion.**
Definierter Einfluß nur auf Übertragstatusbit C und Hilfsübertragstatusbit A. C=1 nur wenn ein (dezimaler) Übertrag ("Borgen") nach AH erfolgt ist; A=1 nur wenn bei der hexadezimalen Subtraktion in AL ein Übertrag ("Borgen") vom niederwertigen Halbbyte erfolgt ist.

ADC	reg,reg	(2 Byte)	
ADC	(mem),reg	(2 Byte)(mnkl)(jj(ii))	
ADC	reg,(mem)	(2 Byte)(mnkl)(jj(ii))	
ADC	AL,FZ	(1 Byte)(gg)	**nur bei AL**
ADC	AX,FZ	(1 Byte)(ggff)	**nur bei AX**
ADC	reg,FZ	(2 Byte)(gg(ff))	
ADC	reg,+FZ	(2 Byte)(ff)	nur Word-Operation
ADC	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)	
ADC	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)	
ADC	WO (mem),+FZ	(2 Byte)(mnkl)(jj(ii))(ff)	

(ADd with Carry)
(Hexadezimale) Addition des rechts vom Komma Stehenden zu dem links vom Komma Stehenden unter gleichzeitiger Addition des Übertragstatusbits C. Ergebnis im links vom Komma Stehenden.
Beeinflussung der Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

ADD	reg,reg	(2 Byte)	
ADD	(mem),reg	(2 Byte)(mnkl)(jj(ii))	
ADD	reg,(mem)	(2 Byte)(mnkl)(jj(ii))	
ADD	AL,FZ	(1 Byte)(gg)	**nur bei AL**
ADD	AX,FZ	(1 Byte)(ggff)	**nur bei AX**
ADD	reg,FZ	(2 Byte)(gg(ff))	

ADD reg,+FZ (2 Byte)(ff) nur Word-Operation
ADD BY (mem),FZ (2 Byte)(mnkl)(jj(ii))(gg)
ADD WO (mem),FZ (2 Byte)(mnkl)(jj(ii))(ggff)
ADD WO (mem),+FZ (2 Byte)(mnkl)(jj(ii))(ff)

(ADDition)
(Hexadezimale) Addition des rechts vom Komma Stehenden zu dem links vom Komma Stehenden. Ergebnis im links vom Komma Stehenden. Beeinflussung der Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

AND reg,reg (2 Byte)
AND (mem),reg (2 Byte)(mnkl)(jj(ii))
AND reg,(mem) (2 Byte)(mnkl)(jj(ii))
AND AL,FZ (1 Byte)(gg) **nur bei AL**
AND AX,FZ (1 Byte)(ggff) **nur bei AX**
AND reg,FZ (2 Byte)(gg(ff))
AND reg,+FZ (2 Byte)(ff) nur Word-Operation
AND BY (mem),FZ (2 Byte)(mnkl)(jj(ii))(gg)
AND WO (mem),FZ (2 Byte)(mnkl)(jj(ii))(ggff)
AND WO (mem),+FZ (2 Byte)(mnkl)(jj(ii))(ff)

Logische UND-Verbindung des rechts vom Komma Stehenden mit dem links vom Komma Stehenden. Als Ergebnis bleiben im links vom Komma Stehenden die Bits nur an den Stellen stehen, an denen sie in **beiden** vorhanden sind (sowohl - als auch).
Die Statusbits für Überlauf (O) und Übertrag (C) werden auf Null gesetzt, das Statusbit für Hilfsübertrag (A) ist nicht definiert, beeinflußt werden die Statusbits für Null (Z), Vorzeichen (S) und Parität (P).

CALL ...

Verzweigung zu Unterprogramm, Aufruf des Unterprogramms: Siehe S. 29. Vor der Verzweigung wird (bei CALL L,... das Befehlsregister CS und dann) der auf den nächsten Befehl weisende Wert von IP an der Spitze des Stapels abgelegt. Kein Einfluß auf das Statusregister.

CALL klmn:fgij (1 Byte = 9A) ij fg mn kl

definiert neues CS:IP.

CALL klmn (1 Byte = E8) mn kl

neues IP im gleichen CS: der Wert von IP des Befehls CALL... selbst vermehrt um drei plus klmn ergibt den IP des Unterprogramms IP(neu) = IP(call) + klmn + 3.
NB.: Hier besteht ein wesentlicher Unterschied zum Mikroassembler von DEBUG: Bei diesem würde klmn **die Adresse des UP** bedeuten, hier wird mit klmn diese Adresse erst berechnet.

CALL /opqr/ (1 Byte = E8) jj ii

neues IP im gleichen CS durch die Adresse /opqr/ festgelegt.
NB.: Im Maschinencode gibt iijj **nicht** diese neue Adresse an, sondern entsprechend dem obigen klmn, wie diese Adresse berechnet wird.

CALL /opqr/ + klmn (1 Byte = E8) jj ii

neues IP im gleichen CS durch die Adresse /opqr/ vermehrt um klmn festgelegt.

CALL ±qr (1 Byte = E8) jj ii

Aufruf des UP, das in der Zeile beginnt, die die um ±qr vermehrte Zeilenzahl des CALL-Befehls hat.

CALL reg (2 Byte)

Inhalt von reg (nur Word-Operationen!) definiert neues IP im gleichen CS.

CALL (mem) (2 Byte)(mnkl)(jj(ii))

Inhalt von mem und mem + 1 (scheinbare Vertauschung!) definiert neues IP im gleichen CS.

CALL L,(mem) (2 Byte)(mnkl)(jj(ii))

das Wort (mem und mem + 1, scheinbare Vertauschung) definiert neues IP, das Word (mem + 2 und mem + 3, scheinbare Vertauschung) das neue CS.
Ab Auflage DOS 3.0 heißt dieser Befehl rückassembliert (der Assembler akzeptiert beide Formen):

CALL FAR (mem) (2 Byte)(mnkl)(jj(ii))

CBW -- (1 Byte)

(Convert Byte to Word)
Vorzeichenverlängerung Byte-Word: das höchste Bit von AL wird in alle Bit von AH kopiert.
Kein Einfluß auf das Statusregister.

CLC -- (1 Byte)

(CLear Carry flag)
Lösche Übertragstatusbit; mache C gleich 0. (Gegenbefehl: STC).

CLD -- (1 Byte)

(CLear Direction flag)
Lösche Direction-Statusbit; mache D gleich 0. D=0 ist der Normalzustand, der beim Beginn des Programmablaufs vorhanden ist.
D=0 heißt: aufwärtszählen = vergrößere bei allen nachfolgenden String-Operationen SI bzw. DI um 1 bzw. (nach Word-Operationen) um 2. (Gegenbefehl: STD; vgl. S. 22/23).

CLI -- (1 Byte)

(CLear Interrupt flag)
Lösche Unterbrechungsstatusbit; mache I gleich 0.
I=0 heißt: es werden keine Unterbrechungsanforderungen der angeschlossenen Hardware akzeptiert. (Gegenbefehl STI; vgl. S. 22/23).

CMC -- (1 Byte)

(CoMplement Carry flag)
Negiere das Übertragstatusbit; wenn C=0, mache C=1; wenn C=1, mache C=0.

CMP	reg,reg	(2 Byte)	
CMP	(mem),reg	(2 Byte)(mnkl)(jj(ii))	
CMP	reg,(mem)	(2 Byte)(mnkl)(jj(ii))	
CMP	AL,FZ	(1 Byte)(gg)	**nur bei AL**

CMP	AL,"a"	(1 Byte)(gg)	**nur bei AL**
CMP	AX,FZ	(1 Byte)(ggff)	**nur bei AX**
CMP	AX,"ab"	(1 Byte)(ggff)	**nur bei AX**
CMP	reg,FZ	(2 Byte)(gg(ff))	
CMP	reg,"a(b)"	(2 Byte)(gg(ff))	
CMP	reg, + FZ	(2 Byte)(ff)	nur Word-Operation
CMP	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)	
CMP	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)	
CMP	WO (mem), + FZ	(2 Byte)(mnkl)(jj(ii))(ff)	

(CoMPare)
Vergleiche durch Subtraktion: das rechts vom Komma Stehende wird von dem links vom Komma Stehenden subtrahiert, dieses aber bleibt (auch) unverändert: das Ergebnis ist **nur** in den Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P) charakterisiert.
Anstelle von Festzahlen können, wie angegeben, in manchen Fällen Zeichen(folgen), unter Anführungszeichen gesetzt, angegeben werden. Wenn es sich dabei um **Zeichenfolgen**, also zwei "Buchstaben", handelt, sind sie hier in der textbezogenen Reihenfolge (also für den Anfang von "Abend" als "Ab", nicht "scheinbar vertauscht" als "bA") einzugeben. Für ein Halbregister (AL, CH usw.) kann natürlich nur ein Zeichen, für ein ganzes Register müssen zwei Zeichen angegeben werden.

CMPSB -- (1 Byte)

(CoMPare String Bytewise)
Vergleiche (wie bei CMP...) durch Subtraktion das Byte DS:SI (SI kann durch den unmittelbar vorausgehenden Befehl SEG..., bzw. CS: usw., mit einem anderen Segmentregister verknüpft werden) mit dem Byte ES:DI (fest; keine Segmentänderung möglich). Ergebnis **nur** in den Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P). Wenn D = 0, addiere anschließend 1 zu SI und DI; wenn D = 1, subtrahiere anschließend 1 von SI und DI (zu D s.S. 22/23).
Diesem Befehl kann der Wiederholungsbefehl REPZ bzw. REPNZ vorausgehen.

CMPSW -- (1 Byte)

(CoMPare String Wordwise)
Vergleiche (wie bei CMP...) durch Subtraktion das Word DS:SI (SI kann durch den unmittelbar vorausgehenden Befehl SEG..., bzw. CS: usw., mit

einem anderen Segmentregister verknüpft werden) mit dem Word ES:DI (fest; keine Segmentänderung möglich). Ergebnis nur in den Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P). Wenn D = 0, addiere anschließend 2 zu SI und DI; wenn D = 1, subtrahiere anschließend 2 von SI und DI (zu D s.S. 22/23). Diesem Befehl kann der Wiederholungsbefehl REPZ bzw. REPNZ vorausgehen.

CWD -- (1 Byte)

(Convert Word to Doubleword)
Vorzeichenverlängerung Wort-Doppelwort: das höchste Bit von AX wird in alle Bits von DX kopiert.
Kein Einfluß auf das Statusregister.

CS: -- (1 Byte) **auch**: SEG CS

(CodeSegment)
Präfix. Ordnet dem nächsten Befehl (und nur diesem) das von der Normalzuweisung abweichende Segmentregister CS zu (vgl. Adressierung S. 25/26).

DAA -- (1 Byte)

(Decimal Adjust for Addition)
Korrektur nach einer Addition zweier ein- bis zweistelliger gepackter BCD-Dezimalzahlen (siehe Befehl AAA), deren Ergebnis in AL steht, so daß dieses (eigentlich durch hexadezimale Addition mit den Befehlen ADD oder ADC entstandene) Ergebnis wiederum als gepackte BCD-Dezimalzahl erscheint. Ist dieses Ergebnis größer als 99, wird das Übertragstatusbit gleich 1 gesetzt.
Definierte Beeinflussung der Statusbits für Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

DAS -- (1 Byte)

(Decimal Adjust for Subtraction)
Korrektur nach einer Subtraktion zweier ein- bis zweistelliger gepackter BCD-Dezimalzahlen (siehe Befehl AAA), deren Ergebnis in AL steht, so daß dieses (eigentlich durch hexadezimale Addition mit den Befehlen SUB oder SBB entstandene) Ergebnis wiederum als gepackte BCD-Dezimalzahl erscheint. Ist dieses Ergebnis kleiner als 0, wird das Übertragsta-

tusbit gleich 1 gesetzt.
Definierte Beeinflussung der Statusbits für Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

DEC	reg	(1 Byte)
DEC	BY (mem)	(2 Byte)(mnkl)(jj(ii))
DEC	WO (mem)	(2 Byte)(mnkl)(jj(ii))

(DECrement)
1 subtrahieren.
Beeinflussung der Statusbits für Überlauf (O), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P) (das Übertragstatusbit C wird **nicht** beeinflußt).

DIV	AL,reg	nur Byte-Operation	(2 Byte)
DIV	AL,(mem)	nur Byte-Operation	(2 Byte)(mnkl)(jj(ii))
DIV	AX,reg	nur Word-Operation	(2 Byte)
DIV	AX,(mem)	nur Word-Operation	(2 Byte)(mnkl)(jj(ii))

Ab DOS 3.0 bleiben rückassembliert die Bezeichnungen AL bzw. AX weg, dafür sind bei (mem) entweder BY (BYTE PTR) oder WO (WORD PTR) vorausgestellt (der Assembler akzeptiert alle drei Formen); also:

DIV	reg	(2 Byte)	
DIV	BY (mem)	(2 Byte)(mnkl)(jj(ii))	oder:
DIV	BYTE PTR (mem)	(2 Byte)(mnkl)(jj(ii))	
DIV	WO (mem)	(2 Byte)(mnkl)(jj(ii))	oder:
DIV	WORD PTR (mem)	(2 Byte)(mnkl)(jj(ii))	

(DIVide)
Division ohne Vorzeichen (das jeweils höchste Bit wird nicht als Vorzeichen interpretiert).
Bei der **Byte**-Operation wird (trotz der Angabe "AL") **AX** durch das in reg bzw. (mem) eingetragene **Byte** dividiert; der Quotient ist anschließend in AL, der Rest in AH. Ist der Quotient größer als FF, wird die Unterbrechungsroutine "Division durch Null" aufgerufen. Es ist also zweckmäßig, vor diesem Befehl festzustellen, daß AH kleiner ist als der Divisor (das in reg bzw. (mem) eingetragene Byte).
Bei der **Word**-Operation wird (trotz der Angabe "AX") das Doppelwort **DX-AX** (vgl. den Befehl CWD) durch das in reg bzw. (mem) eingetrage-

ne Wort dividiert; der Quotient ist anschließend in AX, der Rest in DX. Ist der Quotient größer als FFFF, wird die Unterbrechungsroutine "Division durch Null" aufgerufen. Es ist also zweckmäßig, vor diesem Befehl festzustellen, daß DX kleiner ist als der Divisor (das in reg bzw. (mem) eingetragene Wort).
Die Statusbits sind nicht definiert.

DS: -- (1 Byte) **auch:** SEG DS

(Data-Segment)
Präfix. Ordnet dem nächsten Befehl (und nur diesem) das von der Normalzuweisung abweichende Segmentregister DS zu (vgl. Adressierung S. 25/26).

ES: -- (1 Byte) **auch:** SEG ES

(Extra-Segement)
Präfix. Ordnet dem nächsten Befehl (und nur diesem) das von der Normalzuweisung abweichende Segmentregister ES zu (vgl. Adressierung S. 25/26).

ESC (mem) (2 Byte)(mnkl)(jj(ii))(gg(ff))

(ESCape)
Gib den Inhalt von mem auf den Datenbus. Dieser Befehl ist nur für Arbeiten mit einem Co-Prozessor zu verwenden.

HLT -- (1 Byte)

(HaLT)
Hält den Prozessor an. Nur ein (von außen) eingegebenes RESET kann den Prozessor zur Weiterarbeit veranlassen.

IDIV	AL,reg	nur Byte-Operation	(2 Byte)
IDIV	AL,(mem)	nur Byte-Operation	(2 Byte)(mnkl)(jj(ii))
IDIV	AX,reg	nur Word-Operation	(2 Byte)
IDIV	AX,(mem)	nur Word-Operation	(2 Byte)(mnkl)(jj(ii))

Ab DOS 3.0 bleiben rückassembliert die Bezeichnungen AL bzw. AX weg, dafür sind bei (mem) entweder BY (BYTE PTR) oder WO

(WORD PTR) vorausgestellt (der Assembler akzeptiert alle drei Formen); also:

IDIV	reg	(2 Byte)
IDIV	BY (mem)	(2 Byte)(mnkl)(jj(ii)) oder:
IDIV	BYTE PTR (mem)	(2 Byte)(mnkl)(jj(ii))
IDIV	WO (mem)	(2 Byte)(mnkl)(jj(ii)) oder:
IDIV	WORD PTR (mem)	(2 Byte)(mnkl)(jj(ii))

(Integer DIVision)
Division **mit** Vorzeichen (das jeweils höchste Bit wird als Vorzeichen interpretiert).
Bei der **Byte**-Operation wird (trotz der Angabe "AL") **AX** durch das in reg bzw. (mem) eingetragene **Byte** dividiert; der Quotient ist anschließend in AL, der Rest in AH. Ist der Quotient größer als 7F, wird die Unterbrechungsroutine "Division durch Null" aufgerufen. Es ist also zweckmäßig, vor diesem Befehl festzustellen, daß AH kleiner ist als der Divisor (das in reg bzw. (mem) eingetragene Byte).
Bei der **Word**-Operation wird (trotz der Angabe "AX") das Doppelwort **DX-AX** (vgl. den Befehl CWD) durch das in reg bzw. (mem) eingetragene Wort dividiert; der Quotient ist anschließend in AX, der Rest in DX. Ist der Quotient größer als 7FFF, wird die Unterbrechungsroutine "Division durch Null" aufgerufen. Es ist also zweckmäßig, vor diesem Befehl festzustellen, daß DX kleiner ist als der Divisor (das in reg bzw. (mem) eingetragene Wort).
Die Statusbits sind nicht definiert.

IMUL	AL,reg	nur Byte-Operation	(2 Byte)
IMUL	AL,(mem)	nur Byte-Operation	(2 Byte)(mnkl)(jj(ii))
IMUL	AX,reg	nur Word-Operation	(2 Byte)
IMUL	AX,(mem)	nur Word-Operation	(2 Byte)(mnkl)(jj(ii))

Ab DOS 3.0 bleiben rückassembliert die Bezeichnungen AL bzw. AX weg, dafür sind bei (mem) entweder BY (BYTE PTR) oder WO (WORD PTR) vorausgestellt (der Assembler akzeptiert alle drei Formen); also:

IMUL	reg	(2 Byte)
IMUL	BY (mem)	(2 Byte)(mnkl)(jj(ii)) oder:
IMUL	BYTE PTR (mem)	(2 Byte)(mnkl)(jj(ii))

IMUL WO (mem) (2 Byte)(mnkl)(jj(ii)) oder:
IMUL WORD PTR (mem) (2 Byte)(mnkl)(jj(ii))

(Integer MULtiply)
Multiplikation mit Vorzeichen (das höchste Bit der Operanden wird als Vorzeichen interpetiert).
Bei **Byte**-Operationen steht das Ergebnis in AX, bei **Word-Operationen** in DX-AX (vgl. den Befehl CWD).
Definierte Beeinflussung der Statusbits für Überlauf (O) und Übertrag (C): Sind nach der Operation beide gleich 0, dann heißt dies, daß die Bits von AH bzw. DX nur die Kopien des Vorzeichenbits von AL bzw. AX sind.

INB kl (1 Byte)(kl)
INB DX (1 Byte)

(INput Bytewise)
In das Register AL wird das Byte eingetragen, das auf dem durch das angegebene Byte bzw. durch DX festgelegten Eingabe/Ausgabe-Kanal bereitsteht.
Kein Einfluß auf das Statusregister.

INC reg (1 Byte)
INC BY (mem) (2 Byte)(mnkl)(jj(ii))
INC WO (mem) (2 Byte)(mnkl)(jj(ii))

(INCrement)
1 addieren.
Beeinflußung der Statusbits für Überlauf (O), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität(P), (das Übertragstatusbit wird **nicht** beeinflußt; s. DEC).

INT kl (2 Byte) **außer:**
INT 3 (1 Byte = CC)

(INTerrupt)
Unterbrechungsanforderung.
Vor der Unterbrechung wird zunächst das Statusregister an der Spitze des Stapels gespeichert (als wäre der Befehl PUSHF eingegeben), dann wird CS in den Stapel übertragen (wie PUSH CS) und weiter IP (wie ein Befehl "PUSH IP", den es nicht gibt!).
Das Programm verzweigt dann auf das Unterprogramm, dessen Adresse

vom System so berechnet wird: die im Operanden angegebene Hexadezimalzahl wird mit 4 multipliziert, das am dadurch errechneten Speicherplatz im Segment 0000 stehende Wort in IP, das dort folgende Wort in CS übertragen; Programmfortsetzung an diesem CS:IP.
Im Statusregister werden die Kennzeichen für Unterbrechung (I) und Trace (T) zum Start des Unterprogramms auf Null gesetzt; nach der Rückkehr aus dem Unterprogramm mit Iret ist das Statusregister so wie vor dem INT-Befehl.

INTO -- (1 Byte)

(INTerrupt on Overflow)
Unterbrechungsanforderung bei Überlauf (O). Eine fest eingebaute Routine.
Bei Überlauf (O) = 1 werden im Statusregister die Kennzeichen für Unterbrechung (I) und Trace (T) auf Null gesetzt; nach der Rückkehr aus dem Unterprogramm ist das Statusregister so wie vor dem INTO-Befehl.

INW kl (1 Byte)(kl)
INW DX (1 Byte)

(INput Wordwise)
In das Register AX wird das Word eingetragen, das auf dem durch das angegebene Byte bzw. durch DX festgelegten Eingabe/Ausgabe-Kanal bereitsteht.
Kein Einfluß auf das Statusregister.

IRET -- (1 Byte)

(Interrupt RETurn)
Rückkehr von Programmunterbrechung (letzter Befehl eines durch INT ... aufgerufenen Unterprogramms). Entnimmt dem Stapel der Reihe nach wieder IP, CS, Statusregister.

Sprünge siehe Eingabe von Festzahlen S. 27 und die allgemeinen Ausführungen zu Sprüngen usw. S. 29 ff. Die Sprünge haben keinerlei Einfluß auf das Statusregister.

Bei allen Sprungbefehlen sind folgende **Formen des Operanden**

möglich, die bei den einzelnen Sprungbefehlen nur noch **pauschal mit ARG** zusammengefaßt werden (beim Befehl JMP sind noch weitere Operanden möglich):

J..	kl	(1 Byte)(kl)
J..	±qr	(1 Byte)(kl)
J..	/opqr/+klmn	(1 Byte)(kl)

Die **erste** Darstellung bedeutet, daß zum IP des Befehls J.. kl+2 hinzugezählt werden; dabei wird allerdings das Byte kl als Zahl **mit Vorzeichen** interpretiert, so daß Sprünge nach vorne von 0 bis 7F (dez.: 127) und zurück von 80 (dez.: -128) bis FE (dez.: -2, was allerdings einen "Sprung" zum Sprungbefehl selbst, also eine Endlosschleife bedeutet), möglich sind.
Die **zweite** Darstellung bedeutet einen Sprung um qr **Zeilen** nach vorne (+) oder zurück (-).
Die **dritte** Darstellung bedeutet einen Sprung zum Befehl, der im Programm durch die Marke opqr, eventuell mit hexadezimaler Verschiebung klmn, gekennzeichnet ist.
Im zweiten und dritten Fall rechnet der Assembler die Angaben des Operanden in die erste Form um, so daß also bedingte Sprünge über den bei der ersten Darstellung angegebenen Bereich (über dez. -128 bis +127 Byte) nicht möglich sind. Sollte dieser Bereich aber überschritten werden, **ersetzt** der Assembler den einen Befehl durch eine (kurze) Befehls**folge**, die sozusagen den Sprungweitenbereich auf das ganze Segment ausdehnt, die Bedingung des Sprungbefehls aber nicht beeinflußt. Die durch diese Verlängerung des Programms notwendigen Änderungen des Programms wurden S. 39 besprochen.

JA ARG (s.o. allgemeine Einführung "vor JA")

(Jump if Above)
Sprung bei (C)=0 **und** (Z)=0; nach einem Vergleich: Sprung, wenn das links vom Komma Stehende größer ist als das rechts vom Komma Stehende ohne Vorzeichen (das höchste Bit wird nicht als Vorzeichen gerechnet). Sonst weiterfahren mit dem nächsten Befehl.

JAE ARG identisch mit JNC

(Jump if Above or Equal)

JB ARG identisch mit JC

(Jump if Below)

JBE ARG (s.o. vor JA)

(Jump if Below or Equal)
Sprung bei (C) = 1 oder (Z) = 1; nach einem Vergleich: Sprung, wenn (links) kleiner/gleich (rechts) ohne Vorzeichen.

JC ARG (s.o. vor JA)

(Jump if Carry)
Sprung bei (C) = 1; nach einem Vergleich: Sprung, wenn (links) kleiner (rechts) ohne Vorzeichen.

JCXZ ARG (s.o. vor JA)

(Jump if CX = 0)
Sprung bei CX = 0.

JE ARG identisch mit JZ

(Jump if Equal)

JG ARG (s.o. vor JA)

(Jump if Greater)
Sprung bei (S) = 0 und (Z) = 0; nach einem Vergleich: Sprung, wenn (links) größer (rechts) **mit** Vorzeichen.

JGE ARG (s.o. vor JA)

(Jump if Greater or Equal)
Sprung bei (S) = 0; nach einem Vergleich: Sprung, wenn (links) größer/gleich (rechts) **mit** Vorzeichen.

JL ARG (s.o. vor JA)

(Jump if Less)
Sprung bei (S) ≠ (O); nach einem Vergleich: Sprung, wenn (links) kleiner (rechts) **mit** Vorzeichen.

JLE ARG (s.o. vor JA)

(Jump if Less or Equal)
Sprung bei (S)≠(O) oder (Z)=1; nach einem Vergleich: Sprung, wenn (links) kleiner/gleich (rechts) **mit** Vorzeichen.

JMP klmn:fgij (1 Byte = EA) ij fg mn kl

(JuMP)
unbedingter (langer oder weiter) Sprung zu der durch den Operanden definierten Adresse.
Durch klmn:fgij wird ein neues CS:IP definiert: Sprung dorthin.

JMP klmn (1 Byte = E9) mn kl

neues IP im gleichen CS: der Wert von IP des Befehls JMP... selbst vermehrt um drei plus klmn ergibt den IP der Programmfortsetzung:
IP(neu) = IP(jmp) + klmn + 3
NB.: Hier besteht ein wesentlicher Unterschied zum Assembler von DEBUG: Bei diesem würde klmn die **Adresse der Programmfortsetzung** bedeuten, hier wird mit klmn diese Adresse erst berechnet.

JMP /opqr/ (1 Byte = E9) jj ii

neues IP im gleichen CS durch die Adresse /opqr/ festgelegt.
NB.: Im Maschinencode gibt iijj **nicht** diese neue Adresse an, sondern entsprechend dem obigen klmn, wie diese Adresse berechnet wird.

JMP /opqr/ + kl (1 Byte = E9) jj ii

neues IP im gleichen CS durch die Adresse /opqr/ vermehrt um kl festgelegt.

JMP /opqr/ + klmn (1 Byte = E9) jj ii

neues IP im gleichen CS durch die Adresse /opqr/ vermehrt um klmn festgelegt.

JMP ±qr (1 Byte =E9) jj ii

Fortsetzung des Programms in der Zeile, die die um ±qr vermehrte **Zeilenzahl** des JMP-Befehls hat.

NB.: Obwohl hier große Sprünge möglich sind, ist diese Form ungünstig, wenn eventuell noch Korrekturen gemacht werden müssen; die vorher genannte Form mit Adressen/Marken ist im allgemeinen vorzuziehen.

JMP reg (2 Byte)

Inhalt von reg (nur Word-Operationen!) definiert neues IP im gleichen CS.

JMP (mem) (2 Byte)(mnkl)(jj(ii))

Inhalt von mem und mem + 1 (scheinbare Vertauschung!) definiert neues IP im gleichen CS.

JMP L,(mem) (2 Byte)(klmn)(jj(ii))

Das Wort in mem und mem + 1 (scheinbare Vertauschung!) definiert neues IP, das Wort in mem + 2 und mem + 3 (scheinbare Vertauschung!) das neue CS.
Ab Auflage DOS 3.0 heißt dieser Befehl rückassembliert (der Assembler akzeptiert auch diese Form):

JMP FAR (mem) (2 Byte)(mnkl)(jj(ii))

JMPS ARG (s.o. vor JA)

(JuMP Short)
unbedingter kurzer Sprung zu der durch den Operanden definierten Adresse. Dieser Befehl existiert als solcher im Assembler von DEBUG ab Version 3.0 nicht. Vielmehr wird je nach berechneter Sprungweite dieser Befehl (2 Byte lang) oder der hier durch JMP charakterisierte Befehl (3 Byte lang) verwendet. Falls im Quellenprogramm dieser Befehl verwendet wurde und die Sprungweite zu groß ist, wird dieser Befehl automatisch durch JMP (wie oben) ersetzt. Daraus folgende Änderungen: siehe oben vor JA.

JNA ARG identisch mit JBE

(Jump if Not Above)

JNAE ARG identisch mit JC

(Jump if Not Above or Equal)

JNB ARG identisch mit JNC

(Jump if Not Below)

JNBE ARG identisch mit JA

(Jump if Not Below or Equal)

JNC ARG (s.o. vor JA)

(Jumpf if Not Carry)
Sprung bei (C) = 0; nach einem Vergleich: Sprung, wenn (links) größer/gleich (rechts) ohne Vorzeichen.

JNE ARG identisch mit JNZ

(Jump if Not Equal)

JNG ARG identisch mit JLE

(Jump if Not Greater)

JNGE ARG identisch mit JL

(Jump if Not Greater or Equal)

JNL ARG identisch mit JGE

(Jump if Not Less)

JNLE ARG identisch mit JG

(Jump if Not Less or Equal)

JNO ARG (s.o. vor JA)

Sprung bei (O) = 0 (kein Überlauf).

JNP ARG identisch mit JPO

(Jump if Not Parity)

JNS ARG (s.o. vor JA)

(Jump if Not Sign)
Sprung bei (S) = 0 (positives Vorzeichen).

JNZ ARG (s.o. vor JA)

(Jump if Not Zero)
Sprung bei (Z) = 0; nach einem Vergleich: Sprung, wenn (links) ungleich (rechts) mit/ohne Vorzeichen. Nach dem Befehl TEST: Sprung, wenn das abgefragte Bit vorhanden ist oder **nicht fehlt**, bzw. wenn zumindest eines der abgefragten Bits vorhanden ist oder **nicht fehlt**.

JO ARG (s.o. vor JA)

(Jump if Overflow)
Sprung bei (O) = 1 (Überlauf).

JP ARG (s.o. vor JA)

(Jump if Parity)
Sprung bei (P) = 1, Parität gerade.

JPE ARG identisch mit JP

(Jump if Parity Equal)

JPO ARG (s.o. vor JA)

(Jump if Parity Odd)
Sprung bei (P) = 0, Parität ungerade.

JS ARG (s.o. vor JA)

(Jump on Sign)
Sprung bei (S) = 1 (negatives Vorzeichen).

JZ ARG (s.o. vor JA)

(Jump if Zero)
Sprung bei (Z) = 1; nach einem Vergleich: Sprung, wenn (links) gleich (rechts) mit/ohne Vorzeichen; nach dem Befehl TEST: Sprung, wenn das abgefragte Bit **nicht** vorhanden ist oder **fehlt**, bzw. wenn **alle** angefragten Bits **nicht** vorhanden sind oder **fehlen**.

LAHF -- (1 Byte)

(Load AH from Flags)
Übertrage die acht niederwertigen Statusbit nach AH. In diesen sind die Kennzeichen A, C, S, P und Z enthalten, also außer O alle, die für bedingte Sprünge wichtig sind.
Keine Änderung des Statusregisters. (Gegenbefehl SAHF)

LDS reg,(mem) nur Word-Operation (2 Byte)(mnkl)(jj(ii))

(Load DS und register)
Überträgt das Wort von mem und mem + 1 (scheinbare Vertauschung!) in das angegebene Register und anschließend das Wort von mem + 2 und mem + 3 (scheinbare Vertauschung!) in das Register DS.
Kein Einfluß auf das Statusregister.

LEA reg,(mem) nur Word-Operation (2 Byte)(mnkl)(jj(ii))

(Load Effective Address)
Rechnet die effektive Adresse "mem" aus, also z.B. die Summe BX + DI + klmn, und gibt **diese** in das angegebene Register.
Kein Einfluß auf das Statusregister.

LES reg,(mem) nur Word-Operation (2 Byte)(mnkl)(jj(ii))

(Load DS und register)
Überträgt das Wort von mem und mem + 1 (scheinbare Vertauschung!) in das angegebene Register und anschließend das Wort von mem + 2 und mem + 3 (scheinbare Vertauschung!) in das Register ES.
Kein Einfluß auf das Statusregister.

LOCK -- (1 Byte)

Ein Präfix, das nur beim Gebrauch von Co-Prozessoren nützlich ist: Der

8086/8088 hat während der Dauer des nächsten Befehls die Kontrolle über den Systembus.

LODSB -- (1 Byte)

(LOad String Bytewise)
Kopiert das Byte aus DS:SI (durch den vorausgehenden Befehl SEG... auf ein anderes Segmentregister zu beziehen) nach AL und SI wird bei D=0 um 1 erhöht, bzw. bei D=1 um 1 vermindert. (Der Befehl REP kann vorausgehen, ist hier aber sinnlos) (zu D s.S. 22/23).
Kein Einfluß auf das Statusregister.

LODSW -- (1 Byte)

(LOad String Wordwise)
Kopiert das Wort aus DS:SI (durch den vorausgehenden Befehl SEG... auf ein anderes Segmentregister zu beziehen) nach AX (Vorsicht - scheinbare Vertauschung: das im Speicher an [SI] stehende Byte wird nach AL, das an [SI+1] stehende Byte wird nach AH gebracht) und SI wird bei D=0 um 2 erhöht, bzw. bei D=1 um 2 vermindert. (Der Befehl REP kann vorausgehen, ist hier aber sinnlos) (zu D s.S. 22/23).
Kein Einfluß auf das Statusregister.

LOOP ARG (s.o. vor JA)

CX wird um 1 vermindert; Sprung, wenn dann immer noch CX≠0.
Achtung: Ist vor der durch LOOP festgelegten Schleife CX=0, dann wird die Schleife 10000(hex) = 65536(dez) mal durchlaufen. Sonst genau CX-mal. Oder: Die Schleife wird einmal durchlaufen und dann (CX-1) mal wiederholt. (Vgl. im Gegensatz dazu die Befehle REP..).
Kein Einfluß auf das Statusregister.

LOOPE ARG identisch mit LOOPZ

(LOOP if Equal)

LOOPNE ARG identisch mit LOOPNZ

(LOOP if Not Equal)

LOOPNZ ARG (s.o. vor JA)

(LOOP if Not Zero)
CX wird um 1 vermindert; Sprung, wenn dann immer noch CX≠0 **und** (Z)=0 durch eine vorausgehende Operation. (Zu CX siehe oben bei LOOP). Die Schleife wird durchlaufen, solange z.B. ein vorausgehender Vergleich Ungleichheit ("solange ungleich") ergibt, höchstens jedoch CX-mal.
Kein Einfluß auf das Statusregister.

LOOPZ ARG (s.o. vor JA)

(LOOP if Zero)
CX wird um 1 vermindert; Sprung, wenn dann immer noch
CX≠0 **und** (Z)=1 durch eine vorausgehende Operation. (Zu CX siehe oben bei LOOP). Die Schleife wird durchlaufen, solange z.B. ein vorausgehender Vergleich Gleichheit ("solange gleich") ergibt, höchstens jedoch CX-mal.
Kein Einfluß auf das Statusregister.

MOV	reg,reg	(2 Byte)	
MOV	(klmn),AL	(1 Byte)(mnkl)	**nur für AL**
MOV	(klmn),AX	(1 Byte)(klmn)	**nur für AX** sonst:
MOV	(mem),reg	(2 Byte)(klmn)(jj(ii))	
MOV	AL,(klmn)	(1 Byte)(klmn)	**nur für AL**
MOV	AX,(klmn)	(1 Byte)(klmn)	**nur für AX** sonst:
MOV	reg,(mem)	(2 Byte)(klmn)(jj(ii))	
MOV	AL,FZ	(1 Byte)(gg)	**nur für AL**
MOV	AL,"a"	(1 Byte)(gg)	**nur für AL**
MOV	AX,FZ	(1 Byte)(ggff)	**nur für AX**
MOV	AX,"ab"	(1 Byte)(ggff)	**nur für AX** sonst:
MOV	reg,FZ	(2 Byte)(gg(ff))	
MOV	reg,"a(b)"	(2 Byte)(gg(ff))	
MOV	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)	
MOV	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)	
MOV	(mem),segreg	(2 Byte)(mnkl)(jj(ii))	
MOV	reg,segreg	(2 Byte)	
(MOV	CS,(mem)	**nicht erlaubt!)**	
MOV	DS,(mem)	(2 Byte)(mnkl)(jj(ii))	

MOV	ES,(mem)	(2 Byte)(mnkl)(jj(ii))
MOV	SS,(mem)	(2 Byte)(mnkl)(jj(ii))
(MOV	CS,reg	**nicht erlaubt!**)
MOV	DS,reg	(2 Byte)
MOV	ES,reg	(2 Byte)
MOV	SS,reg	(2 Byte)

(MOVe)
Kopiere das rechts vom Komma Stehende in das links vom Komma Stehende; das rechts Stehende bleibt unverändert.
Zur Eingabe von Zeichen (Buchstaben), unter Anführungszeichen gesetzt, siehe bei Befehl CMP.
Die auch hier naheliegende Form der Eingabe von Festzahlen **+FZ** ist nicht erlaubt!
Kein Einfluß auf das Statusregister.

MOVSB -- (1 Byte)

(MOVe String Bytewise)
Kopiert das **Byte** von DS:SI (SI kann durch den vorausgehenden Befehl SEG... auf ein anderes Register bezogen werden) nach ES:DI (feste Verbindung); anschließend werden bei (D)=0 SI und DI um 1 erhöht, bei (D)=1 um 1 vermindert (zu D s.S. 22/23).
Mit vorausgehendem Befehl REP (der hier gleich REPZ und REPNZ ist) kann mit diesem Befehl ein ganzer Block von Byte übertragen werden.
Kein Einfluß auf das Statusregister.

MOVSW -- (1 Byte)

(MOVe String Wordwise)
Kopiert das **Word** von DS:SI (SI kann durch den vorausgehenden Befehl SEG... auf ein anderes Register bezogen werden) nach ES:DI (feste Verbindung); anschließend werden bei (D)=0 SI und DI um 2 erhöht, bei (D)=1 um 2 vermindert (zu D s.S. 22/23).
Mit vorausgehendem Befehl REP (der hier gleich REPZ und REPNZ ist) kann mit diesem Befehl ein ganzer Block von Byte übertragen werden.
Kein Einfluß auf das Statusregister.

MUL AL,reg nur Byte-Operation (2 Byte)
MUL AL,(mem) nur Byte-Operation (2 Byte)(mnkl)(jj(ii))
MUL AX,reg nur Word-Operation (2 Byte)
MUL AX,(mem) nur Word-Operation (2 Byte)(mnkl)(jj(ii))

Ab DOS 3.0 bleiben rückassembliert die Bezeichnungen AL bzw. AX weg, dafür sind bei (mem) entweder BY (BYTE PTR) oder WO (WORD PTR) vorausgestellt (der Assembler akzeptiert alle drei Formen); also:

MUL reg (2 Byte)
MUL BY (mem) (2 Byte)(mnkl)(jj(ii)) oder:
MUL BYTE PTR (mem) (2 Byte)(mnkl)(jj(ii))
MUL WO (mem) (2 Byte)(mnkl)(jj(ii)) oder:
MUL WORD PTR (mem) (2 Byte)(mnkl)(jj(ii))

(MULtiply)
Multiplikation ohne Vorzeichen (das jeweils höchste Bit wird nicht als Vorzeichen interpretiert).
Bei **Byte**-Operationen steht das Ergebnis in AX, bei **Word**-Operationen in DX-AX (vgl. den Befehl CWD).
Definierte Beeinflussung der Statusbits für Überlauf (O) und Übertrag (C): Sind nach der Operation beide gleich 0, dann heißt dies, daß die Bits von AH bzw. DX nur die Kopien des Vorzeichenbits von AL bzw. AX sind.

NEG reg (2 Byte)
NEG BY (mem) (2 Byte)(mnkl)(jj(ii))
NEG WO (mem) (2 Byte)(mnkl)(jj(ii))

(NEGate)
Das höchste Bit des bezeichneten Registers/Speicherplatzes wird als Vorzeichen interpretiert und die entsprechende Zahl mit vertauschtem Vorzeichen im selben Register/Speicherplatz gebildet (Allgemeiner: das Zweierkomplement wird gebildet).
Beeinflussung der Statusbits für Überlauf(O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

NOP	--	(1 Byte)

(No OPeration)
Keine Operation. (Faktisch identisch mit dem Befehl XCHG AX,AX).

NOT	reg	(2 Byte)
NOT	BY (mem)	(2 Byte)(mnkl)(jj(ii))
NOT	WO (mem)	(2 Byte)(mnkl)(jj(ii))

Logische bitweise Negation; jedes Bit des bezeichneten Registers/Speicherplatzes, das 1 ist, wird durch 0 ersetzt und umgekehrt.
Kein Einfluß auf das Statusregister.

OR	reg,reg	(2 Byte)	
OR	(mem),reg	(2 Byte)(klmn)(jj(ii))	
OR	reg,(mem)	(2 Byte)(klmn)(jj(ii))	
OR	AL,FZ	(1 Byte)(gg)	**nur für AL**
OR	AX,FZ	(1 Byte)(ggff)	**nur für AX**
OR	reg,FZ	(2 Byte)(gg(ff))	
OR	reg, +FZ	(2 Byte)(ff)	nur Wort-Operation
OR	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)	
OR	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)	
OR	WO (mem), +FZ	(2 Byte)(mnkl)(jj(ii))(ff)	

Logisches bitweises Oder; Ergebnis links. Im links vom Komma Stehenden wird ein Bit gleich 1 gesetzt, wenn an dieser Stelle bereits im links vom Komma Stehenden oder auch nur im rechts vom Komma Stehenden (oder eben in beiden) ein Bit gleich 1 vorhanden war.
Die Statusbits für Überlauf (O) und Übertrag (C) werden auf Null gesetzt, das Statusbit für Hilfsübertrag (A) ist nicht definiert, beeinflußt werden die Statusbits für Null (Z), Vorzeichen (S) und Parität (P).

OUTB	kl	(1 Byte)(kl)
OUTB	DX	(1 Byte)

(OUTput Bytewise)
In den durch das angegebene Byte kl bzw. durch DX festgelegten Eingabe/Ausgabe-Kanal wird das Byte eingetragen, das im Register AL bereitsteht.
Kein Einfluß auf das Statusregister.

OUTW	kl	(1 Byte)(kl)
OUTW	DX	(1 Byte)

(OUTput Wordwise)
In den durch das angegebene Byte kl bzw. durch DX festgelegten Eingabe/Ausgabe-Kanal wird das Word eingetragen, das im Register AX bereitsteht.
Kein Einfluß auf das Statusregister.

POP	reg	(1 Byte)
(POP	CS	**nicht erlaubt!**)
POP	DS	(1 Byte)
POP	ES	(1 Byte)
POP	SS	(1 Byte)
POP	(mem)	nur Wort-Operation (2 Byte)(mnkl)(jj(ii))

Das an der "Spitze" des Stapels stehende Word wird in das angegebene Register / den angegebenen Speicherplatz kopiert und anschließend SP um 2 erhöht. (Regel: Was als letztes durch den Befehl PUSH... in den Stapel eingetragen wurde, wird als erstes wieder durch den Befehl POP ... aus dem Stapel entnommen). Siehe S. 41, 44-47.
Kein Einfluß auf das Statusregister.

POPF	--	(1 Byte)

(POP Flags)
Das an der "Spitze" des Stapels stehende Wort wird in das Statusregister kopiert. (Weitere Angaben s. Befehl POP).
Statusregister entsprechend verändert!

PUSH	reg	(1 Byte)
PUSH	segreg	(1 Byte)
PUSH	(mem)	nur Wort-Operation (2 Byte)(mnkl)(jj(ii))

Das im abgegebenen Register/Speicherplatz stehende Wort wird an die "Spitze" des Stapels kopiert (**vorher** wird SP um 2 vermindert). (Weitere Angaben s. Befehl POP).
Kein Einfluß auf das Statusregister.

PUSHF -- (1 Byte)

(PUSH Flags)
Das Statusregister wird an die "Spitze" des Stapels kopiert (vorher wird SP um 2 vermindert). (Weitere Angaben s. Befehl POP).
Kein Einfluß auf das Statusregister.

RCL reg (2 Byte)
RCL BY (mem) (2 Byte)(mnkl)(jj(ii))
RCL WO (mem) (2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma bei der Eingabe.)
(Rotate through Carry Left)
Linksrotieren des angegebenen Registers/Speicherplatzes durch das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Wort nach unten zum Kreis gebogen vor, zwischen das höchste und das niedrigste Bit, also unten, das Übertragstatusbit eingeschoben. Der Befehl RCL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um eine Stelle nach links (gegen den Uhrzeigersinn).
Das Übertragstatusbit übernimmt den Wert des höchsten Bits; ist anschließend an die Ausführung des Befehls der Wert des höchstwertigen Bits ungleich dem Wert des Übertragstatusbits, dann wird das Überlaufstatusbit (O) gleich 1 gesetzt.

RCL reg,CL (2 Byte)
RCL BY (mem),CL (2 Byte)(mnkl)(jj(ii))
RCL WO (mem),CL (2 Byte)(mnkl)(jj(ii))

CL-mal linksrotieren des angegebenen Registers/Speicherplatzes durch das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor, zwischen das höchste und das niedrigste Bit das Übertragstatusbit eingeschoben. Der Befehl RCL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um CL Stellen nach links (gegen den Uhrzeigersinn).
Das Übertragstatusbit übernimmt den Wert des CL-höchsten Bits.

RCR reg (2 Byte)
RCR BY (mem) (2 Byte)(mnkl)(jj(ii))
RCR WO (mem) (2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma.)
(Rotate through Carry Right)
Rechtsrotieren des angegebenen Registers/Speicherplatzes durch das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor, zwischen das höchste und das niedrigste Bit, also unten, das Übertragstatusbit eingeschoben. Der Befehl RCL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um eine Stelle nach rechts (im Uhrzeigersinn).
Das Übertragstatusbit übernimmt den Wert des niedrigsten Bits; ist anschließend an die Ausführung des Befehls der Wert des höchstwertigen Bits ungleich dem Wert des Übertragstatusbits, dann wird das Überlaufstatusbit (O) gleich 1 gesetzt.

RCR reg,CL (2 Byte)
RCR BY (mem),CL (2 Byte)(mnkl)(jj(ii))
RCR WO (mem),CL (2 Byte)(mnkl)(jj(ii))

CL-mal rechtsrotieren des angegebenen Registers/Speicherplatzes durch das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor, zwischen das höchste und das niedrigste Bit das Übertragstatusbit eingeschoben. Der Befehl RCL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um CL Stellen nach rechts (im Uhrzeigersinn).
Das Übertragstatusbit übernimmt den Wert des CL-niedrigsten Bits.

REP -- (1 Byte)

(REPeat)
REP ist ein Präfix: Der nach REP geschriebene String-Befehl MOVSB bzw. MOVSW oder STOSB bzw. STOSW wird CX-mal **ausgeführt**; bei CX=0 also Null-mal (vgl. im Gegensatz dazu die Bedeutung von CX=0 beim Befehl LOOP). Prinzipiell kann REP auch als vorlaufender Befehl vor LODSB bzw. LODSW verwendet werden, ist hier aber im allgemeinen nicht sinnvoll, weil nur das zuletzt geladene Byte bzw. Word in AL bzw. AX verbleibt. Im Zusammenhang mit anderen Befehlen ist REP sinnlos.

REPE -- identisch mit REPZ

(REPeat if Equal)

REPNE -- identisch mit REPNZ

(REPeat if Not Equal)

REPNZ -- (1 Byte)

(REPeat if Not Zero)
Präfix. Der nach REP geschriebene String-Befehl SCASB bzw. SCASW oder CMPSB bzw. CMPSW wird ausgeführt, solange das Statusbit für Null (Z) gleich 0 ist, bis also der im nachfolgenden Stringbefehl ausgeführte Vergleich Gleichheit festgestellt hat, **höchstens jedoch** CX-mal (für CX=0: siehe bei REP): "solange ungleich - höchstens CX-mal". Im Zusammenhang mit anderen Befehlen ist REPNZ sinnlos.

REPZ -- (1 Byte)

(REPeat if Zero)
Präfix. Der nach REP geschriebene String-Befehl SCASB bzw. SCASW oder CMPSB bzw. CMPSW wird ausgeführt, solange das Statusbit für Null (Z) gleich 1 ist, bis also der im nachfolgenden Stringbefehl ausgeführte Vergleich Ungleichheit festgestellt hat, höchstens jedoch CX-mal (für CX=0: siehe bei REP): "solange gleich - höchstens CX-mal". Im Zusammenhang mit anderen Befehlen ist REPZ sinnlos.

RET	--	(1 Byte)	**Hier ist kein Kommentar möglich!**
RET	klmn		(1 Byte)(mnkl)
RET	L		(1 Byte)
RET	L,klmn		(1 Byte)(mnkl)

(Ab DOS 3.0 heißen die beiden letzten Befehle rückassembliert RETF bzw. RETF klmn; der Assembler akzeptiert auch diese Formen.)
(RETurn)
Jeweils letzter Befehl eines Unterprogramms: Rückkehr vom Unterprogramm ins übergeordnete Programm (zu dem auf den aktuellen CALL-Befehl folgenden Befehl). Von der Spitze des Stapels wird der auf den nächsten auszuführenden Befehl verweisende Wert von IP geholt und anschließend SP um 2 erhöht - dann bei RET L.. das für das übergeordnete Programm gültige CS mit anschließender Erhöhung von SP um 2. Wenn im Operanden klmn angegeben ist, wird dieser Wert klmn zusätzlich zu SP addiert; dadurch können Daten, die vor Aufruf des Unterprogramms im Stapel abgelegt wurden, übersprungen werden.

ROL reg (2 Byte)
ROL BY (mem) (2 Byte)(mnkl)(jj(ii))
ROL WO (mem) (2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma.)
(ROtate Left)
Linksrotieren des angegebenen Registers/Speicherplatzes mit Abzweigung in das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor. Der Befehl ROL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um eine Stelle nach links (gegen den Uhrzeigersinn), wobei das höchstwertige Bit zugleich in das Übertragstatusbit und in das niedrigstwertige Bit übertragen wird.
Das Übertragstatusbit übernimmt den Wert des höchsten Bits; ist anschließend an die Ausführung des Befehls der Wert des höchstwertigen Bits ungleich dem Wert des Übertragstatusbits, dann wird das Überlaufstatusbit (O) gleich 1 gesetzt.

ROL reg,CL (2 Byte)
ROL BY (mem),CL (2 Byte)(mnkl)(jj(ii))
ROL WO (mem),CL (2 Byte)(mnkl)(jj(ii))

CL-mal linksrotieren des angegebenen Registers/Speicherplatzes mit Abzweigung in das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor. Der Befehl ROL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um CL Stellen nach links (gegen den Uhrzeigersinn), wobei das höchstwertige Bit jedesmal sowohl in das Übertragstatusbit als auch in das niedrigstwertige Bit übertragen wird. Das Übertragstatusbit übernimmt schließlich den Wert des CL-höchsten Bits.

ROR reg (2 Byte)
ROR BY (mem) (2 Byte)(mnkl)(jj(ii))
ROR WO (mem) (2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma.)
(ROtate Right)
Rechtsrotieren des angegebenen Registers/Speicherplatzes mit Abzweigung in das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor. Der Befehl ROL, wie hier an-

gegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um eine Stelle nach rechts (im Uhrzeigersinn), wobei das niedrigstwertige Bit zugleich in das Übertragstatusbit und das höchstwertige Bit übertragen wird.
Das Übertragstatusbit übernimmt den Wert des niedrigsten Bits; ist anschließend an die Ausführung des Befehls der Wert des höchstwertigen Bits ungleich dem Wert des Übertragstatusbits, dann wird das Überlaufstatusbit (O) gleich 1 gesetzt.

ROR reg,CL (2 Byte)
ROR BY (mem),CL (2 Byte)(mnkl)(jj(ii))
ROR WO (mem),CL (2 Byte)(mnkl)(jj(ii))

CL-mal rechtsrotieren des angegebenen Registers/Speicherplatzes mit Abzweigung in das Statusbit für Übertrag (C). Man stelle sich das Byte bzw. Word nach unten zum Kreis gebogen vor. Der Befehl ROL, wie hier angegeben, bewirkt eine Drehung des eingeschriebenen Zeichenmusters um CL Stellen nach rechts (im Uhrzeigersinn), wobei das niedrigstwertige Bit jedesmal sowohl in das Übertragstatusbit als auch in das höchstwertige Bit übertragen wird.
Das Übertragstatusbit übernimmt den Wert des CL-niedrigsten Bits.

SAHF -- (1 Byte)

(Store AH into Flags)
Überträgt AH in das niederwertige Statusregister (Gegenbefehl zu LAHF, dort nähere Angaben).

SAL -- identisch mit SHL

(Shift Arithmetic Left)

SAR reg (2 Byte)
SAR BY (mem) (2 Byte)(mnkl)(jj(ii))
SAR WO (mem) (2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma.)
(Shift Arithmetic Right)
Rechtsverschieben in das Übertragstatusbit mit Bewahrung des Vorzeichens. Das Zeichenmuster im Register/Speicherplatz wird nach rechts

verschoben; dabei wird das "herausfallende" niedrigwertigste Bit in das Übertragstatusbit übertragen und in das höchstwertige Bit dasjenige nachgeschoben, was vorher schon da war. Ist anschließend der Wert des höchsten Bit ungleich dem Übertragstatusbit, so wird das Überlaufstatusbit (O) gleich 1 gesetzt.

SAR	reg,CL	(2 Byte)
SAR	BY (mem),CL	(2 Byte)(mnkl)(jj(ii))
SAR	WO (mem),CL	(2 Byte)(mnkl)(jj(ii))

CL-mal rechtsverschieben in das Übertragstatusbit mit Bewahrung des Vorzeichens. Das Zeichenmuster im Register/Speicherplatz wird nach rechts verschoben; dabei wird das "herausfallende" niedrigwertigste Bit jedesmal in das Übertragstatusbit übertragen und in das höchstwertige Bit dasjenige nachgeschoben, was vorher schon da war.

SBB	reg,reg	(2 Byte)
SBB	(mem),reg	(2 Byte)(mnkl)(jj(ii))
SBB	reg,(mem)	(2 Byte)(mnkl)(jj(ii))
SBB	AL,FZ	(1 Byte)(gg) **nur bei AL**
SBB	AX,FZ	(1 Byte)(ggff) **nur bei AX**
SBB	reg,FZ	(2 Byte)(gg(ff))
SBB	reg, +FZ	(2 Byte)(ff) nur Word-Operation
SBB	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)
SBB	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)
SBB	WO (mem), +FZ	(2 Byte)(mnkl)(jj(ii))(ff)

(SuBtract with Borrow)
(Hexadezimale) Subtraktion des rechts vom Komma Stehenden von dem links vom Komma Stehenden unter gleichzeitiger Subtraktion des Übertragstatusbits. Ergebnis im links vom Komma Stehenden.
Beeinflussung der Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

SCASB	--	(1 Byte)

(SCAn String Bytewise)
Vergleicht durch Subtraktion das Byte in ES:DI (fest) mit AL; anschließend wird bei (D)=0 DI um 1 erhöht, bei (D)=1 um 1 verringert (zu D s.S. 22/23).

Nur der Vergleich beeinflußt die Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

SCASW -- (1 Byte)

(SCan String Wordwise)
Vergleicht durch Subtraktion das Word in ES:DI (fest) mit AX; anschließend wird bei (D) = 0 DI um 2 erhöht, bei (D) = 1 um 2 verringert (zu D s.S. 22/23).
Nur der Vergleich beeinflußt die Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

SEG	CS	(1 Byte)	**auch**: CS:
SEG	DS	(1 Byte)	**auch**: DS:
SEG	ES	(1 Byte)	**auch**: ES:
SEG	SS	(1 Byte)	**auch**: SS:

(SEGment ...)
Präfix. Ordnet dem nächsten Befehl (und nur diesem) ein bestimmtes, von der Normalzuweisung abweichendes Segmentregister zu (vgl. Adressierung S. 25/26).

SHL	reg	(2 Byte)
SHL	BY (mem)	(2 Byte)(mnkl)(jj(ii))
SHL	WO (mem)	(2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma.)
(SHift Left)
Linksverschieben des angegebenen Registers/Speicherplatzes in das Statusbit für Übertrag (C). Der Befehl SHL, wie hier angegeben, bewirkt eine Verschiebung des eingeschriebenen Zeichenmusters um eine Stelle nach links, wobei das höchstwertige Bit in das Übertragstatusbit übertragen und in das niedrigstwertige Bit eine Null nachgeschoben wird.
Das Übertragstatusbit übernimmt den Wert des höchsten Bits; ist anschließend an die Ausführung des Befehls der Wert des höchstwertigen Bits ungleich dem Wert des Übertragstatusbits, dann wird das Überlaufstatusbit (O) gleich 1 gesetzt.

SHL reg,CL (2 Byte)
SHL BY (mem),CL (2 Byte)(mnkl)(jj(ii))
SHL WO (mem),CL (2 Byte)(mnkl)(jj(ii))

CL-mal linksverschieben des angegebenen Registers/Speicherplatzes in das Statusbit für Übertrag (C). Der Befehl SHL, wie hier angegeben, bewirkt eine Verschiebung des eingeschriebenen Zeichenmusters um CL Stellen nach links, wobei das höchstwertige Bit jedesmal in das Übertragstatusbit übertragen und in das niedrigstwertige Bit eine Null nachgeschoben wird.
Das Übertragstatusbit übernimmt schließlich den Wert des CL-höchsten Bits.

SHR reg (2 Byte)
SHR BY (mem) (2 Byte)(mnkl)(jj(ii))
SHR WO (mem) (2 Byte)(mnkl)(jj(ii))

(Ab DOS 3.0 ist rückassembliert bei allen Formen dieses Befehls ",1" angehängt; der Assembler akzeptiert diese Eins mit Komma.)
(SHift Right)
Rechtsverschieben des angegebenen Registers/Speicherplatzes in das Statusbit für Übertrag (C). Der Befehl SHR, wie hier angegeben, bewirkt eine Verschiebung des eingeschriebenen Zeichenmusters um eine Stelle nach rechts, wobei das niedrigstwertige Bit in das Übertragstatusbit übertragen wird und in das höchstwertige Bit eine Null nachgeschoben wird.
Das Übertragstatusbit übernimmt den Wert des niedrigsten Bits; ist anschließend an die Ausführung des Befehls der Wert des höchstwertigen Bits ungleich dem Wert des Übertragstatusbits, dann wird das Überlaufstatusbit (O) gleich 1 gesetzt.

SHR reg,CL (2 Byte)
SHR BY (mem),CL (2 Byte)(mnkl)(jj(ii))
SHR WO (mem),CL (2 Byte)(mnkl)(jj(ii))

CL-mal rechtsverschieben des angegebenen Registers/Speicherplatzes in das Statusbit für Übertrag (C). Der Befehl SHL, wie hier angegeben, bewirkt eine Verschiebung des eingeschriebenen Zeichenmusters um CL Stellen nach rechts, wobei das niedrigstwertige Bit jedesmal in das Übertragstatusbit übertragen und in das höchstwertige Bit eine Null nachgeschoben wird.
Das Übertragstatusbit übernimmt schließlich den Wert des CL-niedrigsten Bits.

SS: -- (1 Byte) **auch: SEG SS**

Präfix. Ordnet dem nächsten Befehl (und nur diesem) das von der Normalzuweisung abweichende Segmentregister SS zu (vgl. Adressierung S. 25/26).

STC -- (1 Byte)

(SeT Carry flag)
Macht das Übertragstatusbit gleich 1. Gegenbefehl zu CLC.

STD -- (1 Byte)

(SeT Direction flag)
Macht das Direction-Statusbit gleich 1. Dies heißt für DI und SI aller **nachfolgenden** String-Befehle: abwärtszählen! Gegenbefehl zu CLD.

STI -- (1 Byte)

(SeT Interrupt flag)
Macht das Unterbrechungsstatusbit gleich 1. Unterbrechungsanforderungen werden wieder akzeptiert. Gegenbefehl zu CLI.

STOSB -- (1 Byte)

(STOre String Bytewise)
Kopiert das Byte von AL nach ES:DI (fest!); anschließend wird bei (D)=0 DI um 1 erhöht, bzw. bei (D)=1 um 1 erniedrigt (zu D s.S. 22/23).
Als vorausgehender Befehl ist REP (hier gleich REPNZ und REPZ) möglich.
Kein Einfluß auf das Statusregister.

STOSW -- (1 Byte)

(STOre String Wordwise)
Kopiert das Word von AX nach ES:DI (fest!); anschließend wird bei (D)=0 DI um 2 erhöht, bzw. bei (D)=1 um 2 erniedrigt (zu D s.S. 22/23).
Als vorausgehender Befehl ist REP (hier gleich REPNZ und REPZ) möglich.
Kein Einfluß auf das Statusregister.

SUB	reg,reg	(2 Byte)
SUB	(mem),reg	(2 Byte)(mnkl)(jj(ii))
SUB	reg,(mem)	(2 Byte)(mnkl)(jj(ii))
SUB	AL,FZ	(1 Byte)(gg) **nur bei AL**
SUB	AX,FZ	(1 Byte)(ggff) **nur bei AX**
SUB	reg,FZ	(2 Byte)(gg(ff))
SUB	reg, +FZ	(2 Byte)(ff) nur Word-Operation
SUB	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)
SUB	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)
SUB	WO (mem), +FZ	(2 Byte)(mnkl)(jj(ii))(ff)

(SUBtract)
(Hexadezimale) Subtraktion des rechts vom Komma Stehenden von dem links vom Komma Stehenden. Ergebnis im links vom Komma Stehenden. Beeinflussung der Statusbits für Überlauf (O), Übertrag (C), Hilfsübertrag (A), Null (Z), Vorzeichen (S) und Parität (P).

TEST	reg,reg	(2 Byte)
TEST	reg,(mem)	(2 Byte)(mnkl)(jj(ii))
TEST	reg,FZ	(2 Byte)(gg(ff))
TEST	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)
TEST	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg(ff))

Die Form +FZ (Angabe nur eines Byte mit Verlängerung des Vorzeichens) ist hier nicht erlaubt!
Vergleich durch logisches UND (siehe Befehl AND) - aber **allein** die Statusregister werden wie folgt beeinflußt: Überlauf- (O) und Übertragstatusbit (C) werden auf Null gesetzt; das Hilfsübertragstatusbit (A) ist nicht definiert; beeinflußt werden die Statusbit für Null (Z) (an keiner Stelle ist sowohl im rechts vom Komma Stehenden als auch in dem links links vom Komma Stehenden ein Bit gleich 1: ergibt (Z) = 1), für Vorzeichen (S) (das höchstwertige Bit in beiden ist gleich 1 ergibt (S) = 1), und Parität (P).

WAIT	--	(1 Byte)

Unterbricht die Operation des Prozessors, solange am Prozessor-Eingang "TEST" ein L-Pegel liegt; nur bei Arbeiten mit Co-Prozessoren als Programmbefehl sinnvoll.

XCHG	reg,AX	(1 Byte)	**nur mit AX**, sonst:
XCHG	reg,reg	(2 Byte)	
XCHG	reg,(mem)	(2 Byte)(mnkl)(jj(ii))	

(eXCHanGe)
Austausch der Inhalte der angegebenen Register/Speicherplätze. Kein Einfluß auf das Statusregister.

XLAT	--	(1 Byte)

(eXchange by transLATion)
Das Byte von DS:(BX+AL) wird nach AL kopiert. Typisch für Zeichenübersetzung: in DS:BX beginnt die Übersetzungstabelle, aus der jeweils das AL-te Zeichen nach AL übernommen wird.

XOR	reg,reg	(2 Byte)	
XOR	(mem),reg	(2 Byte)(mnkl)(jj(ii))	
XOR	reg,(mem)	(2 Byte)(mnkl)(jj(ii))	
XOR	AL,FZ	(1 Byte)(gg)	**nur bei AL**
XOR	AX,FZ	(1 Byte)(ggff)	**nur bei AX**
XOR	reg,FZ	(2 Byte)(gg(ff))	
XOR	reg,+FZ	(2 Byte)(ff)	nur Word-Operation
XOR	BY (mem),FZ	(2 Byte)(mnkl)(jj(ii))(gg)	
XOR	WO (mem),FZ	(2 Byte)(mnkl)(jj(ii))(ggff)	
XOR	WO (mem),+FZ	(2 Byte)(mnkl)(jj(ii))(ff)	

(eXclusive OR)
Logisches Entweder-Oder. Das Ergebnis steht in dem links vom Komma angegebenen Register/Speicherplatz. Im Ergebnis steht jeweils nur an derjenigen Stelle ein Bit gleich 1, an der **entweder** im einen **oder** im andern ein Bit gleich 1 stand; an der Stelle, an der in beiden Operanden der gleiche Wert stand, steht als Ergebnis Null.
Die Statusregister werden wie folgt beeinflußt: Überlauf- (O) und Übertragstatusbit (C) werden auf Null gesetzt; das Hilfsübertragstatusbit (A) ist nicht definiert; beeinflußt werden die Statusbits für Null (Z), für Vorzeichen (S) und Parität (P).

VII. Die wichtigsten Interruptfunktionen

Der Operand des Befehls INT ist ein Byte lang; so wäre es prinzipiell möglich, 256 verschiedene Interruptfunktionen festzulegen. Offensichtlich aber sollten einige dieser Funktionen für den Programmierer reserviert bleiben; andererseits sind einige der vorprogrammierten Funktionen so ineinander verkettet, daß es nicht zweckmäßig schien, mit diesen mehrere Werte des Operanden zu belegen - diese sind jeweils in eine Gruppe zusammengefaßt und ihre Unterscheidung erfolgt im Register AH: dadurch sind 256 Unterfunktionen möglich - beim Interrupt INT 21 sind ab DOS-Version 3.0 etwa neunzig bereits verwendet. Alle diese Interruptfunktionen hier aufzuzeigen, scheint nicht sinnvoll; so sollen nur die "wichtigsten" angeführt werden.

Zunächst zwei Interruptfunktionen, die keine Unterfunktionen haben:

INT 3 (1 Byte = CC)

Programmunterbrechung.
Dieser Interrupt ist sehr praktisch, solange ein Programm im Teststadium ist: Da der Maschinencode nur ein Byte lang ist (= CC), kann man ihn unter DEBUG anstelle des ersten Bytes eines jeden beliebigen Befehls setzen, der Programmablauf wird an dieser Stelle gestoppt; zusätzlich werden bei dieser Programmunterbrechung (unter DEBUG) die Werte aller Register am Bildschirm angezeigt, so daß der Zustand des Prozessors an dieser Stelle des Programms kontrolliert werden kann.

INT 20 (2 Byte = CD 20)

Programmabbruch.
Das Programm wird abgebrochen, die vor dem Start des

Programms gültige Systemanfrage erscheint auf dem Bildschirm (unter DEBUG zusätzlich die Anzeige: "Programm normal beendet")

Die am häufigsten gebrauchte Interruptfunktion, INT 21, und einige ihrer Unterfunktionen.

Die Interruptfunktion INT 21 erfordert, da sie Unterfunktionen besitzt, zumindest eine Spezifikation im Register AH; je nach Art der so spezifizierten Unterfunktion können noch zusätzliche Angaben in anderen Registern erforderlich sein, so daß dieser Interrupt immer nur am Ende einer Befehlsfolge aufgerufen werden kann.

Im folgenden werden diese zusätzlichen Angaben durch die Befehle MOV... dargestellt; es ist natürlich möglich, daß die erforderlichen Angaben in den entsprechenden Registern (AH, DX usw.) durch das Programm vorher errechnet werden und somit die MOV..-Befehle überflüssig sind. Die betroffenen Register müssen jedenfalls vor dem Befehl INT 21 festgelegt sein.

```
MOV     AH,1
INT     21
```

Eingabe von der Tastatur; das eingegebene Zeichen erscheint auf dem Bildschirm (in der Literatur als "Echo" bezeichnet), sein Hexadezimalcode wird in AL wiedergegeben. Wird CTRL-Break eingegeben, so wird das Programm abgebrochen (in der Literatur: "mit Erkennen von CTRL-Break").

Die Tastatureingabe wird zunächst vom System "gepuffert", d.h. zwischengespeichert, so daß während eines Programmablaufs im allgemeinen bis etwa 16 Zeichen eingegeben werden können, ohne daß das System unmittelbar darauf reagiert. Wenn im Programmablauf dann die beiden Befehle MOV AH,1 und INT 21 auftauchen, übernimmt das

System das **zuerst** eingegebene Zeichen aus dem Puffer. Sollte das System nur auf das Zeichen reagieren, das an dieser Stelle des Programms eingegeben wird, muß man folgende Befehlsreihe verwenden:

```
MOV    AX,C01
INT    21
```

AH = 0C: Löschen des Tastaturpuffers und Ausführen der Funktion, die in AL angegeben ist (die ansonsten in AH angegeben werden müßte), hier also, weil AL = 01: Eingabe von der Tastatur mit "Echo".
Alle zwischenzeitlich (und vielleicht versehentlich) eingegebenen Zeichen werden im Tastaturpuffer gelöscht. Das System **wartet** auf die (Neu-) Eingabe eines Zeichens.

```
MOV    AH,7
INT    21
```

Eingabe von der Tastatur, ohne "Echo" und ohne Erkennen Erkennen von CTRL-Break (s.o.): Der Hexadezimalcode des eingegebenen Zeichens wird in AL wiedergegeben.

```
MOV    AX,C07
INT    21
```

Warten auf die Eingabe von der Tastatur entsprechend der Funktion AH = 7. (s. Bemerkung zu Funktion MOV AX,C01...)

```
MOV    AH,8
INT    21
```

Wie Funktion AH = 7, aber **mit** Erkennen von CTRL-Break.

```
MOV   AX,C08
INT   21
```

Warten auf die Eingabe von der Tastatur entsprechend der Funktion AH = 8. (s. Bemerkung zu Funktion MOV AX,C01...)

```
MOV   AH,A        (bzw. MOV AX,C0A, s.u.)
MOV   DX,klmn     (bzw. MOV DX,/.../)
INT   21
```

Eingabe einer Zeichenkette von der Tastatur in einen Puffer. Dieser Puffer muß durch den Programmierer reserviert werden und muß folgende Form haben: Er beginnt in DS:DX, also im Speicherplatz DX=klmn des Datensegments DS; das **erste** Byte wird beim Programmieren festgelegt und gibt die Höchst-Anzahl der in diesen Puffer eingebbaren Zeichen an (durch diese Festlegung wird vermieden, daß bei der Eingabe einer zu langen Zeichenkette ein eventuell nachfolgendes Programmstück überschrieben wird; die Interruptfunktion gestattet nicht, mehr Zeichen einzugeben); das **zweite** Byte gibt automatisch nach der Eingabe einer Zeichenkette an, wieviele Zeichen **tatsächlich** eingegeben wurden; ab dem Byte **DX+2** werden diese Zeichen "gepuffert", eingeschrieben; die Zeichenkette endet immer mit dem Code für Zeilenschaltung (= Enter) 0D. Insgesamt müssen also für einen solchen Puffer "Höchst-Anzahl plus drei" Byte reserviert werden.

Ein Beispiel für die Verwendung dieses Interrupts in unserem Assembler:

```
0045 DB      15 #17;0#
     .....
     (PUSH   CS
     POP     DS)
     MOV     DX,/45/
     MOV     AH,A        (bzw. MOV AX,C0A)
     INT     21
```

Sollte eine versehentliche Tastatureingabe vor diesem Interrupt auf alle Fälle vermieden werden, kann auch hier die Funktion für Löschen des Tastaturpuffers mitverwendet werden: Anstelle von MOV AH,A wird MOV AX,C0A geschrieben (s.o. bei Eingabe eines Zeichens von der Tastatur: Funktion AH = 1).

```
MOV  AH,9
MOV  DX,klmn            (bzw. MOV DX,/.../)
INT  21
```

Ausgabe einer Zeichenkette auf dem Bildschirm (und, wenn der Drucker zugeschaltet ist, zugleich auf dem Drucker). Die Zeichenkette, die im Speicherplatz klmn des Datensegments, also in DS:DX beginnt, wird bis zum Zeichen "$" bzw. hex 24 auf dem Bildschirm ausgegeben; es ist also wichtig, die Zeichenkette mit "$" abzuschließen, da sonst auch die folgenden Zeichen des Speichers ausgegeben werden, bis irgendwo "zufällig" ein Byte 24 gefunden wird.

Ein Beispiel für die Verwendung dieses Interrupts in unserem Assembler:

```
0034 DB    "Dieser Text wird ausgegeben" D A 24
0035 DB    "Dieser anschließend in einer neuen Zeile.$"
     .....
     (PUSH      CS
     POP        DS)
     MOV        AH,9
     MOV        DX,/34/
     INT        21
     MOV        DX,/35/
     INT        21
```

Die mit 0034 markierte DB-Zeile endet mit den Byte 0D (= Wagenrücklauf, bzw. Versetzen des Schreibzeigers - Cursors, Positionsanzeigers - auf dem Bildschirm an den Zeilenbeginn), 0A (= Zeilenvorschub, bzw. Versetzen des Schreibzeigers in die nächste Zeile) und 24 (="$", Ende der auszugebenden Zeichenreihe); die mit 0035 markierte Zeile nur mit "$", es erfolgt kein "Wagenrücklauf + Zeilenvorschub". Vor dem zweiten Interrupt INT 21 muß AH nicht neu festgelegt werden, weil diese Funktion den Wert von AH nicht verändert.

Eine weitere, fast ebenso reichgestaltete Interruptfunktion ist **INT 10**. Von dieser nur ein Beispiel:

```
MOV   AL,..   in AL Hex-Code des auszugebenden Zeichens
MOV   CX,..   in CX, wie oft dieses Zeichen auszugeben ist
MOV   BH,0    Bildschirmseite (für "Einsteiger" immer = 0)
MOV   BL,..   Attribut-Byte, s.u.
MOV   AH,9
INT   10
```

Die erste Zeile kann natürlich unter entsprechenden Vorausset-

zungen durch LODSB ersetzt werden. In CX muß angegeben werden, wie oft dieses eine Zeichen ausgegeben werden soll (meist wohl CX = 1). Dritte Zeile: das System verwaltet mehrere "Bildschirmseiten"; diesbezüglich kann hier nur auf weiterführende Literatur verwiesen werden - für den "Einsteiger" ist immer BH = 0. Die Attribut-Byte sind für Schwarz-Weiß-Adapter und Farb-Grafik-Adapter des Bildschirms unterschiedlich differenziert. Für beide gilt:

BL = 07 normales Zeichen	BL = 70 invertiertes Zeichen
BL = 0F erhöhte Intensität	BL = 78 invert. erhöht. Int.
BL = 87 blinkend normal	BL = F0 blinkend invertiert

(BL = 8F blinkend erhöhte Intensität: dem Beschauer kaum zuzumuten!)

Die dritte und vierte Zeile im angegebenen Beispiel können normalerweise zusammengefaßt werden in z.B.

MOV BX,87 (Seite 0, blinkend normal),

da in dieser Form BH immer gleich Null gesetzt und nur BL, wie angegeben, festgelegt wird.

VIII. Umrechnung von Dezimal- in Hexadezimalzahlen und umgekehrt

Im Bereich der Dezimalzahlen von 0 bis 255 verwendet man am besten die Tabelle auf den nächsten Seiten. - Diese Tabelle gibt zusätzlich an, wie unter gewissen Voraussetzungen negative Dezimalzahlen als positiv scheinende Hexadezimalzahlen (und umgekehrt) dargestellt werden können. Bei den "kurzen" Sprungbefehlen ist nur **ein** Byte als Operand möglich und dieses Byte wird als "Hexadezimalzahl mit Vorzeichen (VZ)" interpretiert: die größtmögliche positive Hex-Zahl ist dabei 7F (= dezimal 127), die kleinste negative Zahl 80 (= dezimal -128). Siehe dazu die Vorbemerkung vor dem Befehl JA in Abschnitt VI.

Allgemein werden Dezimalzahlen in die entsprechenden Hexadezimalzahlen durch ein Verfahren umgerechnet, das man als "Resteverwertung" bezeichnen könnte: Man dividiert die vorgegebene Dezimalzahl durch 16 und schreibt den **Rest** in hexadezimaler Form an (wenn also der Rest z.B. dezimal 14 wäre, schreibt man diesen als E an). Dann dividiert man das **Ergebnis** der Division (den Quotienten) wieder durch 16, und schreibt den **Rest** an die nächsthöhere Stelle der bisher erhaltenen Hexadezimalzahl an (im Beispiel vor das E); diesen Schritt wiederholt man so lange, bis das Ergebnis der neuerlichen Division durch 16 den Wert Null ergibt und der noch verbleibende Rest in hexadezimaler Form an die höchste Stelle der bisher erhaltenen Hexadezimalzahl geschrieben werden kann.

Die Dezimalzahl 63440 soll umgerechnet werden:

```
63440 : 16  = 3965
 154
  104
   80
    0              Rest = 0dez = 0hex    ──────────┐
 3965 : 16  = 247                                  │
   76                                              │
  125                                              │
   13              Rest = 13dez = Dhex ─────────┐  │
  247 : 16  = 15                                │  │
   87                                           │  │
    7              Rest = 7dez = 7hex  ──────┐  │  │
   15 : 16  = 0                              ↓  ↓  ↓
   15              Rest = 15dez = Fhex ──→ F 7  D  0
```

Die Umrechnung von Hexadezimalzahlen in Dezimalzahlen ist etwas einfacher. Man multipliziert die am weitesten rechts stehende Ziffer als Dezimal-Zahl geschrieben mit 1, die zweite von rechts als Dezimal-Zahl geschrieben mit 16, die dritte von rechts als Dezimal-Zahl geschrieben mit 16x16, also mit 256, die vierte mit 16x16x16, also mit 4096 usw.

Die Hexadzimalzahl DE7F soll als Dezimalzahl dargestellt werden:

F	15x1	=	15
7	7x16	=	112
E	14x256	=	3584
D	13x4096	=	53248
	Summe		56959

Darstellung von Hexadezimalziffern im Binärcode:

hex	binär	hex	binär
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Die Umrechnung ist einfach: Jede Ziffer der Hexadezimalzahl wird durch die entsprechende Vierergruppe des Binärcodes ersetzt und umgekehrt.

Tabelle der Kennzeichen (Flags) und ihrer "Werte" in DEBUG

Kennzeichen	1	0
O = Overflow, Überlauf (Ja/Nein)	OV	NV
D = Direction, Richtung (Abzieh./Erhöh.)	DN	UP
I = Interrupt, Unterbechung (erlaubt/nicht erl.)	EI	DI
S = Sign, Vorzeichen (negativ/positiv)	NG	PL
Z = Zero, Null (Ja/Nein)	ZR	NZ
A = Auxilary Carry, Hilfsübertrag (J/N)	AC	NA
P = Parität, Anzahl der Einsen (gerade/unger.)	PE	PO
C = Carry, Übertrag (Ja/Nein)	CY	NC

Notizen

Sachregister

Notizen

Umrechnungstabelle dez/hex

hex	dezimal mit VZ	dezimal ohne VZ
00	+0	0
01	+1	1
02	+2	2
03	+3	3
04	+4	4
05	+5	5
06	+6	6
07	+7	7
08	+8	8
09	+9	9
0A	+10	10
0B	+11	11
0C	+12	12
0D	+13	13
0E	+14	14
0F	+15	15
10	+16	16
11	+17	17
12	+18	18
13	+19	19
14	+20	20
15	+21	21
16	+22	22
17	+23	23
18	+24	24
19	+25	25
1A	+26	26
1B	+27	27
1C	+28	28
1D	+29	29
1E	+30	30
1F	+31	31

hex	dezimal mit VZ	dezimal ohne VZ
20	+32	32
21	+33	33
22	+34	34
23	+35	35
24	+36	36
25	+37	37
26	+38	38
27	+39	39
28	+40	40
29	+41	41
2A	+42	42
2B	+43	43
2C	+44	44
2D	+45	45
2E	+46	46
2F	+47	47
30	+48	48
31	+49	49
32	+50	50
33	+51	51
34	+52	52
35	+53	53
36	+54	54
37	+55	55
38	+56	56
39	+57	57
3A	+58	58
3B	+59	59
3C	+60	60
3D	+61	61
3E	+62	62
3F	+63	63

hex	dezimal mit VZ	ohne VZ	hex	dezimal mit VZ	ohne VZ
40	+64	64	60	+96	96
41	+65	65	61	+97	97
42	+66	66	62	+98	98
43	+67	67	63	+99	99
44	+68	68	64	+100	100
45	+69	69	65	+101	101
46	+70	70	66	+102	102
47	+71	71	67	+103	103
48	+72	72	68	+104	104
49	+73	73	69	+105	105
4A	+74	74	6A	+106	106
4B	+75	75	6B	+107	107
4C	+76	76	6C	+108	108
4D	+77	77	6D	+109	109
4E	+78	78	6E	+110	110
4F	+79	79	6F	+111	111
50	+80	80	70	+112	112
51	+81	81	71	+113	113
52	+82	82	72	+114	114
53	+83	83	73	+115	115
54	+84	84	74	+116	116
55	+85	85	75	+117	117
56	+86	86	76	+118	118
57	+87	87	77	+119	119
58	+88	88	78	+120	120
59	+89	89	79	+121	121
5A	+90	90	7A	+122	122
5B	+91	91	7B	+123	123
5C	+92	92	7C	+124	124
5D	+93	93	7D	+125	125
5E	+94	94	7E	+126	126
5F	+95	95	7F	+127	127

hex	dezimal	
	mit VZ	ohne VZ
80	-128	128
81	-127	129
82	-126	130
83	-125	131
84	-124	132
85	-123	133
86	-122	134
87	-121	135
88	-120	136
89	-119	137
8A	-118	138
8B	-117	139
8C	-116	140
8D	-115	141
8E	-114	142
8F	-113	143
90	-112	144
91	-111	145
92	-110	146
93	-109	147
94	-108	148
95	-107	149
96	-106	150
97	-105	151
98	-104	152
99	-103	153
9A	-102	154
9B	-101	155
9C	-100	156
9D	-99	157
9E	-98	158
9F	-97	159

hex	dezimal	
	mit VZ	ohne VZ
A0	-96	160
A1	-95	161
A2	-94	162
A3	-93	163
A4	-92	164
A5	-91	165
A6	-90	166
A7	-89	167
A8	-88	168
A9	-87	169
AA	-86	170
AB	-85	171
AC	-84	172
AD	-83	173
AE	-82	174
AF	-81	175
A0	-80	176
B1	-79	177
B2	-78	178
B3	-77	179
B4	-76	180
B5	-75	181
B6	-74	182
B7	-73	183
B8	-72	184
B9	-71	185
BA	-70	186
BB	-69	187
BC	-68	188
BD	-67	189
BE	-66	190
BF	-65	191

hex	dezimal		hex	dezimal	
	mit VZ	ohne VZ		mit VZ	ohne VZ
C0	-64	192	E0	-32	224
C1	-63	193	E1	-31	225
C2	-62	194	E2	-30	226
C3	-61	195	E3	-29	227
C4	-60	196	E4	-28	228
C5	-59	197	E5	-27	229
C6	-58	198	E6	-26	230
C7	-57	199	E7	-25	231
C8	-56	200	E8	-24	232
C9	-55	201	E9	-23	233
CA	-54	202	EA	-22	234
CB	-53	203	EB	-21	235
CC	-52	204	EC	-20	236
CD	-51	205	ED	-19	237
CE	-50	206	EE	-18	238
CF	-49	207	EF	-17	239
D0	-48	208	F0	-16	240
D1	-47	209	F1	-15	241
D2	-46	210	F2	-14	242
D3	-45	211	F3	-13	243
D4	-44	212	F4	-12	244
D5	-43	213	F5	-11	245
D6	-42	214	F6	-10	246
D7	-41	215	F7	-9	247
D8	-40	216	F8	-8	248
D9	-39	217	F9	-7	249
DA	-38	218	FA	-6	250
DB	-37	219	FB	-5	251
DC	-36	220	FC	-4	252
DD	-35	221	FD	-3	253
DE	-34	222	FE	-2	254
DF	-33	223	FF	-1	255

ASCII-Hexadezimalcode für Zeichen

	0	1	2	3	4	5	6	7
0	NUL 0	16	32	0 48	@ 64	P 80	' 96	p 112
1	1	DC1 17	! 33	1 49	A 65	Q 81	a 97	q 113
2	2	DC2 18	" 34	2 50	B 66	R 82	b 98	r 114
3	♥ 3	DC3 19	# 35	3 51	C 67	S 83	c 99	s 115
4	♦ 4	DC4 20	$ 36	4 52	D 68	T 84	d 100	t 116
5	♣ 5	§ 21	% 37	5 53	E 69	U 85	e 101	u 117
6	♠ 6	22	& 38	6 54	F 70	V 86	f 102	v 118
7	BEL 7	23	' 39	7 55	G 71	W 87	g 103	w 119
8	BS 8	CAN 24	(40	8 56	H 72	X 88	h 104	x 120
9	HT 9	25	) 41	9 57	I 73	Y 89	i 105	y 121
A	LF 10	26	* 42	: 58	J 74	Z 90	j 106	ź 122
B	VT 11	ESC 27	+ 43	; 59	K 75	[91	k 107	{ 123
C	FF 12	28	, 44	< 60	L 76	\ 92	l 108	¦ 124
D	CR 13	29	- 45	= 61	M 77	] 93	m 109	} 125
E	SO 14	30	. 46	> 62	N 78	^ 94	n 110	~ 126
F	SI 15	31	/ 47	? 63	O 79	_ 95	o 111	127

	8	9	A	B	C	D	E	F
0	Ç 128	É 144	á 160	176	└ 192	╨ 208	α 224	≡ 240
1	ü 129	æ 145	í 161	▒ 177	┴ 193	╤ 209	β 225	± 241
2	é 130	Æ 146	ó 162	▓ 178	┬ 194	╥ 210	Γ 226	≥ 242
3	â 131	ô 147	ú 163	│ 179	├ 195	╙ 211	π 227	≤ 243
4	ä 132	ö 148	ñ 164	┤ 180	─ 196	╘ 212	Σ 228	⌠ 244
5	à 133	ò 149	Ñ 165	╡ 181	┼ 197	╒ 213	σ 229	⌡ 245
6	å 134	û 150	ª 166	╢ 182	╞ 198	╓ 214	µ 230	÷ 246
7	ç 135	ù 151	º 167	╖ 183	╟ 199	╫ 215	τ 231	≈ 247
8	ê 136	ÿ 152	¿ 168	╕ 184	╚ 200	╪ 216	Φ 232	° 248
9	ë 137	Ö 153	⌐ 169	╣ 185	╔ 201	┘ 217	Θ 233	∙ 249
A	è 138	Ü 154	¬ 170	║ 186	╩ 202	┌ 218	Ω 234	· 250
B	ï 139	¢ 155	½ 171	╗ 187	╦ 203	█ 219	δ 235	√ 251
C	î 140	£ 156	¼ 172	╝ 188	╠ 204	▄ 220	∞ 236	^{n} 252
D	ì 141	¥ 157	¡ 173	╜ 189	═ 205	▌ 221	φ 237	^{2} 253
E	Ä 142	₧ 158	« 174	╛ 190	╬ 206	▐ 222	∈ 238	■ 254
F	Å 143	ƒ 159	» 175	┐ 191	╧ 207	▀ 223	∩ 239	255

Notizen

MS-DOS Taschentabelle für IBM-PC und Kompatible



* Alle MS-DOS Befehle kurz erklärt und mit einem Beispiel untermauert
* Geordnet nach Funktionen
* Für alle MS-DOS Versionen incl. MS-DOS 3.3 und MS-DOS 4.0

Die Abmessungen haben wir so gewählt (Größe: 9 x 17 cm), daß das Heftchen in der Tasche des Oberhemdes Platz hat.

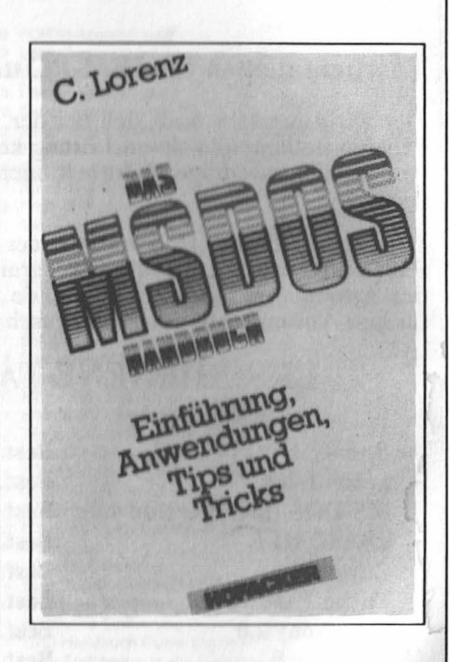
So steht die Information zu jeder Zeit griffbereit zur Verfügung.

MS-DOS Taschentabelle (Heftchen allein)
Best.Nr. 272 nur DM 9,80

MS-DOS Taschentabelle und Diskette mit den Beispielprogrammen
Best.Nr. 2722 nur DM 29,80

MS-DOS Handbuch

Für Version 3.2, 3.3 und 4.0



MS-DOS ist das am meisten installierte Betriebssystem für Personal Computer weltweit. Noch niemals in der Geschichte wurden so viele Exemplare von einem einzigen Betriebssystem verkauft. Bis Ende 1989 dürften ca. 20 Millionen Rechner mit MS-DOS ausgerüstet sein. Dieses Buch vermittelt Ihnen praktisch alles, was Sie als Anwender dieses erfolgreichen Betriebssystems wissen sollten. Auch derjenige, der sich intensiver mit der Programmierung unter MS-DOS befassen will, findet eine Fülle von Informationen, Tips und Hinweisen. So wird z.B. ausführlich auf die Festplattenbenutzung und auf die Interpretation von Fehlermeldungen eingegangen. Viele praktische Programmbeispiele, die Sie selbst nachvollziehen können, runden das Werk ab.

Best.Nr. 133 (Buch alleine) DM 29,80
Best.Nr. 1332 (Disk u. Buch) DM 79,00

Weitere intessante Bücher von Hofacker:

BÜCHER in deutscher Sprache

Best.-Nr.	Titel	Preis/DM
1	Transistor Berechnungs- und Bauanleitungsbuch-1	29,80
2	Transistor Berechnungs- und Bauanleitungsbuch-2	19,80
3	Elektronik im Auto	9,80
4	IC-Handbuch, TTL, CMOS, Linear	19,80
5	IC-Datenbuch, TTL, CMOS, Linear	9,80
6	IC-Schaltungen, TTL, CMOS, Linear	19,80
7	Elektronik Schaltungen	19,80
8	IC-Bauanleitungs-Handbuch	19,80
9	Feldeffekttransistoren	9,80
10	Elektronik und Radio	19,80
13	HEH, Hobby Elektronik Handbuch	9,80
16	CMOS Teil 1, Einführung, Entwurf,Schaltbeispiele	19,80
18	CMOS Teil 3, Entwurf und Schaltbeispiele	19,80
19	IC-Experimentier Handbuch	19,80
20	Operationsverstärker	19,80
21	Digitaltechnik Grundkurs	19,80
22	Mikroprozessoren, Eigenschaften und Aufbau	5,00
23	Elektronik Grundkurs, Kurzlehrgang Elektronik	9,80
24	Programmieren in Maschinensprache Z80,II	5,00
25	68000 Microcomputer Einführung	39,00
26	Mikroprozessor, Teil 2	5,00
27	BASIC-M für 6800/09/68000 (Motorola)	5,00
28	Lexikon u. Wörterbuch f.Eleltr.u.Mikroporzessor	29,80
29	Hardware Handbook	49,00
31	57 Praktische Programme in BASIC	5,00
32	ATARI BASIC, für Selbststudium und Praxis	39,00
66	Profit steigern mit dBASE III	29,80
102	Mathematische + Wissenschaftliche Prog.i.BASIC	29,80
103	Oszillographen-Handbuch	19,80
105	MEVA-Handbuch	29,80
108	Rund um den Spectrum (Progr.,Tips und Tricks)	5,00
109	6502 Microcomputer Programmierung	5,00
111	Programmieren mit TRS-80 (GENIE)	29,80
112	PASCAL-Programmier-Handbuch	9,80
113	BASIC-Programmier-Handbuch (mit BASIC.Kurs)	5,00
114	Der Microcomputer im Kleinbetrieb	39,80
116	Einführung 16-Bit	29,80
118	Programmieren in Maschinensprache mit dem 6502	5,00
119	Programmieren in Maschinensprache Z-80,Teil 1	9,80
120	Anwenderprogramme für TRS-80 und GENIE	29,80
121	Microsoft BASIC-Handbuch	5,00
122	BASIC für Fortgeschrittene	5,00
123	IEC-Bus Handbuch	19,80
124	Progr. in Maschinensprache mit dem Commodore-64	5,00
127	Einführung i.d. Microcomputer-Progr. mit 6800	49,00
128	Programmieren mit dem CBM	29,80
130	Programmbeispiele für CBM	9,80
133	Das MS-DOS Handb.(inkl.Vers.4.0)	29,80
137	FORTH-Grundlagen, Einführung, Beispiele	5,00
140	Progr. i.BASIC u.Maschinencode mit dem ZX-81	5,00
141	Progr.f.VC-20 (Spiele, Utilities, Erweiterungen)	5,00
143	35 Programme für den ZX-81	5,00
144	33 Programme für den ZX-Spectrum	5,00
145	64 Programme für den Commodore-64	5.00
146	Hardware Erweiterungen für den Commodore-64	5,00
147	Beherrschen Sie Ihren Commodore-64	5,00
148	Programmierhandbuch für SHARP	5,00
149	Programme für TI99/4A	5,00
187	Mehr als 29 Programme für den Commodore-64	5,00
188	Statistic in BASIC	39,00
189	6502 Maschinenspr.Beisp. Commodore-64	5,00
194	Das C-Anwenderhandbuch	29,80
200	FORTH-Anwendungen	29,80
202	UNIX-Grundlagen und Anwendungen	39,00
204	Grafik und Ton mit Commodore-64	5,00
205	Das große Spielebuch für ATARI,T.2	29,80
210	Superprogramme für IBM-PC	29,80
212	Geschäftsprogramme für Commodore-64	5,00
213	Technische Gleichungssysteme in BASIC	49,00
216	Wordstar für Füchse	24,80
217	Künstliche Intelligenz	5,00
219	Profit & Produktivität mit Framework	29,80
220	Tabellenkalkulation für blutige Laien	19,80
221	SYMPHONY-Anwendungen	29,80
222	Praktische Anwendungen mit dem Sinclair QL	5,00
223	MODULA-2 Anwender Handbuch	9,80
224	Anwenderprogramme für APPLE II c, und II e	19,80
226	Modem und Hacker-Handbuch	19,80
227	Die große Starparade	5,00
228	Das große Minispione Baubuch	29,80
229	Praktische Einführung in LISP	29,80
230	MSX - Tips und Tricks	5,00
231	8088/8086 Maschinensprache	29,80
232	PROLOG-Handbuch	29,80
233	PROLOG-Anwendungen	29,80
234	LOTUS 1-2-3 für Fortgeschrittene	29,80
235	GOETHE-Utilities	29,80
239	OPEN ACCESS II für Anwender	39,00
240	PROLOG - Eine praktische Einführung	29,80
241	Stapeljobs	9,80
243	Turbo C-Grundkurs	29,80
244	Desk Top Publishing HB(Fontasy)	19,80
245	Praktische Einführung in C	29,80
246	Praktische Einführung in Turbo Pascal 4.0	9,80
247	CAD-Handbuch Curve Digitizer	39,00
248	Prakt.Einführung in die Fraktale Geom.	39,00
250	MS-DOS und WINDOWS	29,80
252	Z-80 Referenzkarte	5,00
253	Public Domain u.Shareware HB, Teil 1	9,80
254	Public Domain u.Shareware HB, Teil 2	9,80
256	GW-BASIC Handbuch	19,80
257	GW-BASIC Schnellkurs	9,80
258	Turbo BASIC Schnellkurs	9,80
259	PC/AT Service + Selbstbau HB	39,00
260	Laserset-Handbuch	39,00
261	Small C-Schnellkurs	9,80
262	Clipper Schnellkurs	9,80
263	Ventura Schnellkurs	9,80
264	FORTH Schnellkurs	9,80
266	Laserjet-Handbuch	29,80
267	Barcode Schnellkurs	9,80
270	BASIC-Taschentabelle	5,00
271	LOTUS 1-2-3 Taschentabelle	9,80
272	MS-DOS 4.0 Taschentabelle	9,80
273	dBASEIII + Taschentabelle	9,80
274	Clipper-Taschentabelle	9,80
275	Turbo PASCAL 5.0 Taschentabelle	9,80
276	C-64 Taschentabelle	5,00
277	SYMPHONY 2.0 Taschentabelle	9,80
278	WORD 4.0 Taschentabelle	9,80
279	FRAMEWORK II/III Taschentabelle	9,80
282	Ein Assembler für Einsteiger	19,80
315	Grafik Buch aus Public Domain und Shareware	19,80
8029	Z-80 Assembler-Handbuch	9,80

8086/8088 Assembler für Einsteiger

Einführung in die Programmierung mit Assembler

für alle Personalcomputer nach dem MS-DOS Standard

Die Programmierung in Assembler ist entgegen vielen Behauptungen auch für Anfänger und Einsteiger sehr interessant. Gerade das maschinennahe Programmieren schafft ein grundlegendes Verständnis für die Funktion eines Computers. In Assembler geschriebene Programme sind schneller als alle anderen, in Hochsprachen geschriebenen Programme und benötigen darüber hinaus auch noch weniger Speicherplatz. Dieses Buch liefert Ihnen einen praxisnahen und problemlosen Einstieg in die 8086/8088 Assembler Programmierung. Beispielprogramme untermauern den Lernprozess. Ganz besonders interessant ist, daß es zu diesem Buch einen Assembler gibt, mit dem Sie die Beispiele im Buch nachvollziehen und später Ihre eigenen Programme entwickeln können. Eine Begleitdiskette enthält neben diesem Assembler auch die Beispielprogramme aus diesem Buch.


ISBN 3-88963-282-3

